Pro Git

Scott Chacon*

2011-07-13

*Ce fichier PDF est la traduction française du livre Pro Git. Il est publié sous license Creative Commons Attribution-Non Commercial-Share Alike 3.0. J'espère que vous l'apprécierez, qu'il vous permettra d'apprendre à utiliser Git et que vous aiderez Apress en achetant le livre original sur Amazon : http://tinyurl.com/amazonprogit

Table des matières

1 Démarrage rapide **1**

1.1 À propos de la gestion de version . . . 1

1.1.1 Les systèmes de gestion de version locaux . . . 1

1.1.2 Les systèmes de gestion de version centralisés . . . 2

1.1.3 Les systèmes de gestion de version distribués . . . 3

1.2 Une rapide histoire de Git . . . 4

1.3 Rudiments de Git . . . 4

1.3.1 Des instantanés, pas des différences . . . 4

1.3.2 Presque toutes les opérations sont locales . . . 5

1.3.3 Git gère l'intégrité . . . 6

1.3.4 Généralement, Git ne fait qu'ajouter des données . . . 6

1.3.5 Les trois états . . . 6

1.4 Installation de Git . . . 7

1.4.1 Installation depuis les sources . . . 8

1.4.2 Installation sur Linux . . . 8

1.4.3 Installation sur Mac . . . 9

1.4.4 Installation sur Windows . . . 9

1.5 Paramétrage à la première utilisation de Git . . . 9

1.5.1 Votre identité . . . 10

1.5.2 Votre éditeur de texte . . . 10

1.5.3 Votre outil de différences . . . 11

1.5.4 Vérifier vos paramètres . . . 11

1.6 Obtenir de l'aide . . . 11

1.7 Résumé . . . 12

2 Les bases de Git **13**

2.1 Démarrer un dépôt Git . . . 13

2.1.1 Initialisation d'un dépôt Git dans un répertoire existant . . . 13

2.1.2 Cloner un dépôt existant . . . 14

2.2 Enregistrer des modifications dans le dépôt . . . 14

2.2.1 Vérifier l'état des fichiers . . . 15

2.2.2 Placer de nouveaux fichiers sous suivi de version . . . 16

2.2.3 Indexer des fichiers modifiés . . . 16

2.2.4 Ignorer des fichiers . . . 18

2.2.5 Inspecter les modifications indexées et non indexées . . . 19

2.2.6 Valider vos modifications . . . 21

2.2.7 Éliminer la phase d'indexation . . . 23
2.2.8 Effacer des fichiers . . . 23
2.2.9 Déplacer des fichiers . . . 24
2.3 Visualiser l'historique des validations . . . 25
2.3.1 Limiter la longueur de l'historique . . . 29
2.3.2 Utiliser une interface graphique pour visualiser l'historique . . . 30
2.4 Annuler des actions . . . 31
2.4.1 Modifier le dernier commit . . . 31
2.4.2 Désindexer un fichier déjà indexé . . . 32
2.4.3 Réinitialiser un fichier modifié . . . 33
2.5 Travailler avec des dépôts distants . . . 34
2.5.1 Afficher les dépôts distants . . . 34
2.5.2 Ajouter des dépôts distants . . . 35
2.5.3 Récupérer et tirer depuis des dépôts distants . . . 35
2.5.4 Pousser son travail sur un dépôt distant . . . 36
2.5.5 Inspecter un dépôt distant . . . 36
2.5.6 Retirer et déplacer des branches distantes . . . 37
2.6 Balisage . . . 38
2.6.1 Lister vos balises . . . 38
2.6.2 Créer des balises . . . 38
2.6.3 Les balises annotées . . . 39
2.6.4 Les balises signées . . . 39
2.6.5 Les balises légères . . . 40
2.6.6 Vérifier des balises . . . 41
2.6.7 Baliser après coup . . . 41
2.6.8 Partager les balises . . . 42
2.7 Trucs et astuces . . . 43
2.7.1 Auto-Complétion . . . 43
2.7.2 Les alias Git . . . 44
2.8 Résumé . . . 45

3 Les branches avec Git **47**
3.1 Ce qu'est une branche . . . 47
3.2 Brancher et fusionner : les bases . . . 52
3.2.1 Le branchement de base . . . 52
3.2.2 Les bases de la fusion . . . 55
3.2.3 Conflits de fusion . . . 57
3.3 Gestion de branches . . . 59
3.4 Travailler avec les branches . . . 61
3.4.1 Branches au long cours . . . 61
3.4.2 Les branches de sujet . . . 62
3.5 Les branches distantes . . . 63
3.5.1 Pousser vers un serveur . . . 65
3.5.2 Suivre les branches . . . 67
3.5.3 Effacer des branches distantes . . . 68
3.6 Rebaser . . . 68

3.6.1 Les bases . . . 68
3.6.2 Rebasages plus intéressants . . . 70
3.6.3 Les dangers de rebaser . . . 72
3.7 Résumé . . . 74

4 Git sur le serveur **75**
4.1 Protocoles . . . 75
4.1.1 Protocole local . . . 76
Avantages . . . 76
Inconvénients . . . 77
4.1.2 Protocole SSH . . . 77
Avantages . . . 77
Inconvénients . . . 78
4.1.3 Protocole Git . . . 78
Avantages . . . 78
Inconvénients . . . 78
4.1.4 Protocole HTTP/S . . . 78
Avantages . . . 79
Inconvénients . . . 80
4.2 Installation de Git sur un serveur . . . 80
4.2.1 Copie du dépôt nu sur un serveur . . . 80
4.2.2 Petites installations . . . 81
Accès SSH . . . 81
4.3 Génération des clefs publiques SSH . . . 82
4.4 Mise en place du serveur . . . 83
4.5 Accès public . . . 85
4.6 GitWeb . . . 86
4.7 Gitosis . . . 88
4.8 Le daemon Git . . . 93
4.9 Git hébergé . . . 95
4.9.1 GitHub . . . 95
4.9.2 Création d'un compte utilisateur . . . 95
4.9.3 Création d'un nouveau dépôt . . . 96
4.9.4 Import depuis Subversion . . . 98
4.9.5 Ajout des collaborateurs . . . 98
4.9.6 Votre projet . . . 99
4.9.7 Duplication de projets . . . 100
4.9.8 Résumé sur GitHub . . . 101
4.10 Résumé . . . 101

5 Git distribué **103**
5.1 Développements distribués . . . 103
5.1.1 Gestion centralisée . . . 103
5.1.2 Mode du gestionnaire d'intégration . . . 104
5.1.3 Mode dictateur et ses lieutenants . . . 105
5.2 Contribution à un projet . . . 106

5.2.1 Guides pour une validation . . . 106
5.2.2 Cas d'une petite équipe privée . . . 108
5.2.3 Équipe privée importante . . . 114
5.2.4 Petit projet public . . . 117
5.2.5 Grand projet public . . . 121
5.2.6 Résumé . . . 124
5.3 Maintenance d'un projet . . . 124
5.3.1 Travail dans des branches thématiques . . . 124
5.3.2 Application des patchs à partir d'e-mail . . . 125
Application d'un patch avec apply . . . 125
Application d'un patch avec am . . . 125
5.3.3 Vérification des branches distantes . . . 128
5.3.4 Déterminer les modifications introduites . . . 129
5.3.5 Intégration des contributions . . . 130
Modes de fusion . . . 130
Gestions avec nombreuses fusions . . . 131
Gestion par rebasage et sélection de commit . . . 133
5.3.6 Balisage de vos publications . . . 135
5.3.7 Génération d'un nom de révision . . . 136
5.3.8 Préparation d'une publication . . . 136
5.3.9 Shortlog . . . 137
5.4 Résumé . . . 137

6 Utilitaires Git 139
6.1 Sélection des versions . . . 139
6.1.1 Révisions ponctuelles . . . 139
6.1.2 Empreinte SHA courte . . . 139
6.1.3 QUELQUES MOTS SUR SHA-1 . . . 140
6.1.4 Références de branches . . . 141
6.1.5 Raccourcis RefLog . . . 141
6.1.6 Références passées . . . 142
6.1.7 Plages de commits . . . 144
Double point . . . 144
Emplacements multiples . . . 145
Triple point . . . 146
6.2 Mise en attente interactive . . . 146
6.2.1 Mettre en attente des fichiers . . . 147
6.2.2 Patches de Staging . . . 149
6.3 La remise . . . 150
6.3.1 Remiser votre travail . . . 151
6.3.2 Créer une branche depuis une remise . . . 153
6.4 Réécrire l'historique . . . 153
6.4.1 Modifier la dernière consignation . . . 154
6.4.2 Modifier plusieurs messages de consignation . . . 154
6.4.3 Réordonner les commits . . . 156
6.4.4 Rassembler des commits . . . 157

6.4.5 Diviser un commit . . . 158
6.4.6 L'option nucléaire : filter-branch . . . 159
Supprimer un fichier de chaque commit . . . 159
Faire d'un sous-répertoire la nouvelle racine . . . 159
Modifier globalement l'adresse mail . . . 160
6.5 Deboguer avec Git . . . 160
6.5.1 Fichier annoté . . . 160
6.5.2 La recherche dichotomique . . . 161
6.6 Sous-modules . . . 163
6.6.1 Démarrer un sous-module . . . 164
6.6.2 Cloner un projet avec des sous-modules . . . 166
6.6.3 Superprojects . . . 168
6.6.4 Les problèmes avec les sous-modules . . . 168
6.7 Fusion de sous-arborescence . . . 170
6.8 Résumé . . . 172

7 Personnalisation de Git 173
7.1 Configuration de Git . . . 173
7.1.1 Configuration de base d'un client . . . 174
core.editor . . . 174
commit.template . . . 174
core.pager . . . 175
user.signingkey . . . 175
core.excludesfile . . . 176
help.autocorrect . . . 176
7.1.2 Couleurs dans Git . . . 176
color.ui . . . 176
color.* . . . 177
7.1.3 Outils externes de fusion et de différence . . . 177
7.1.4 Formatage and espaces blancs . . . 180
core.autocrlf . . . 180
core.whitespace . . . 181
7.1.5 Configuration du serveur . . . 182
receive.fsckObjects . . . 182
receive.denyNonFastForwards . . . 182
receive.denyDeletes . . . 182
7.2 Attributs Git . . . 183
7.2.1 Fichiers binaires . . . 183
Identification des fichiers binaires . . . 183
Comparaison de fichiers binaires . . . 184
7.2.2 Expansion des mots-clés . . . 186
7.2.3 Export d'un dépôt . . . 188
export-ignore . . . 188
export-subst . . . 189
7.2.4 Stratégies de fusion . . . 189
7.3 Crochets Git . . . 190

7.3.1 Installation d'un crochet . . . 190
7.3.2 Crochets côté client . . . 190
Crochets de traitement de validation . . . 190
Crochets de gestion e-mail . . . 191
Autres crochets côté client . . . 191
7.3.3 Crochets côté serveur . . . 192
pre-receive et post-receive . . . 192
update . . . 192
7.4 Exemple de politique gérée par Git . . . 193
7.4.1 Crochets côté serveur . . . 193
Application d'une politique de format du message de validation . . . 193
Mise en place d'un système d'ACL par utilisateur . . . 195
Application des poussées en avance rapide . . . 197
7.4.2 Crochets côté client . . . 199
7.5 Résumé . . . 202

8 Git et les autres systèmes 203
8.1 Git et Subversion . . . 203
8.1.1 git svn . . . 203
8.1.2 Installation . . . 204
8.1.3 Démarrage . . . 205
8.1.4 Valider en retour sur le serveur Subversion . . . 206
8.1.5 Tirer des modifications . . . 207
8.1.6 Le problème avec les branches Git . . . 209
8.1.7 Les embranchements dans Subversion . . . 210
Créer une nouvelle branche SVN . . . 210
8.1.8 Basculer de branche active . . . 210
8.1.9 Les commandes Subversion . . . 211
L'historique dans le style Subversion . . . 211
Les annotations SVN . . . 212
L'information sur la serveur SVN . . . 212
Ignorer ce que Subversion ignore . . . 213
8.1.10 Résumé sur Git-Svn . . . 213
8.2 Migrer sur Git . . . 214
8.2.1 Importer . . . 214
8.2.2 Subversion . . . 214
8.2.3 Perforce . . . 216
8.2.4 Un outil d'import personnalisé . . . 218
8.3 Résumé . . . 223

9 Les trippes de Git 225
9.1 Plomberie et porcelaine . . . 225
9.2 Les objets Git . . . 226
9.2.1 Objets Arbre . . . 228
9.2.2 Objets Commit . . . 231
9.2.3 Stockage des objets . . . 233

9.3 Références Git 235
9.3.1 La branche HEAD 236
9.3.2 Étiquettes 237
9.3.3 Références distantes 238
9.4 Fichiers groupés 239
9.5 Les références spécifiques 242
9.5.1 Publier une référence spécifique 244
9.5.2 Supprimer des références 244
9.6 Protocoles de transfert 244
9.6.1 Le protocole stupide 245
9.6.2 Le Protocole Intelligent 247
Téléverser des données 247
Télécharger des Données 248
9.7 Maintenance et Récupération de Données 249
9.7.1 Maintenance 249
9.7.2 Récupération de données 250
9.7.3 Supprimer des objets 253
9.8 Résumé 256

Chapitre 1

Démarrage rapide

Ce chapitre traite du démarrage rapide avec Git. Nous commencerons par expliquer les bases de la gestion de version, puis nous parlerons de l'installation de Git sur votre système et finalement comment le paramétrer pour commencer à l'utiliser. À la fin de ce chapitre vous devriez en savoir assez pour comprendre pourquoi on parle beaucoup de Git, pourquoi vous devriez l'utiliser et vous devriez en avoir une installation prête à l'emploi.

1.1 À propos de la gestion de version

Qu'est-ce que la gestion de version et pourquoi devriez-vous vous en soucier ? Un gestionnaire de version est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment. Dans les exemples de ce livre, nous utiliserons des fichiers sources de logiciel comme fichiers sous gestion de version, bien qu'en réalité on puisse l'utiliser avec pratiquement tous les types de fichiers d'un ordinateur.

Si vous êtes un dessinateur ou un développeur web, et que vous voulez conserver toutes les versions d'une image ou d'une mise en page (ce que vous souhaiteriez assurément), un système de gestion de version (VCS en anglais pour Version Control System) est un outil qu'il est très sage d'utiliser. Il vous permet de ramener un fichier à un état précédent, ramener le projet complet à un état précédent, comparer les changements au cours du temps, voir qui a modifié quelque chose qui pourrait causer un problème, qui a introduit un problème et quand, et plus encore. Utiliser un VCS signifie aussi généralement que si vous vous trompez ou que vous perdez des fichiers, vous pouvez facilement revenir à un état stable. De plus, vous obtenez tous ces avantages avec une faible surcharge de travail.

1.1.1 Les systèmes de gestion de version locaux

La méthode commune pour la gestion de version est généralement de recopier les fichiers dans un autre répertoire (peut-être avec un nom incluant la date dans le meilleur des cas). Cette méthode est la plus commune parce que c'est la plus simple, mais c'est aussi la moins fiable. Il est facile d'oublier le répertoire dans lequel vous êtes et d'écrire accidentellement dans le mauvais fichier ou d'écraser des fichiers que vous vouliez conserver.

Pour traiter ce problème, les programmeurs ont développé il y a longtemps des VCSs locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier (voir figure

1-1).

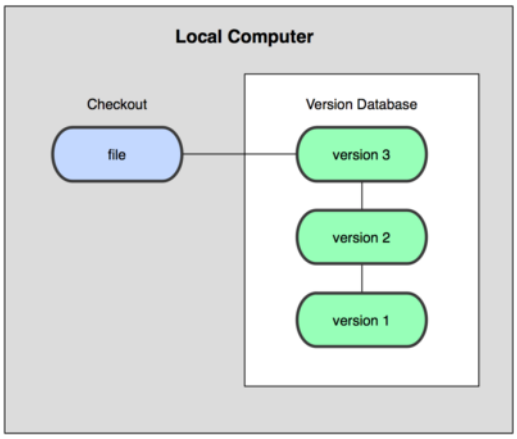

Figure 1.1: *Diagramme des systèmes de gestion de version locaux.*

Un des systèmes les plus populaires était RCS, qui est encore distribué avec de nombreux systèmes d'exploitation aujourd'hui. Même le système d'exploitation populaire Mac OS X inclut le programme rcs lorsqu'on installe les outils de développement logiciel. Cet outil fonctionne en conservant des ensembles de patch (c'est-à-dire la différence entre les fichiers) d'une version à l'autre dans un format spécial sur disque ; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

1.1.2 Les systèmes de gestion de version centralisés

Le problème majeur que les gens rencontrent est qu'ils ont besoin de collaborer avec des développeurs sur d'autres ordinateurs. Pour traiter ce problème, les systèmes de gestion de version centralisés (CVCS en anglais pour *Centralized Version Control Systems*) furent développés. Ces systèmes tels que CVS, Subversion, et Perforce, mettent en place un serveur central qui contient tous les fichiers sous gestion de version, et des clients qui peuvent extraire les fichiers de ce dépôt central. Pendant de nombreuses années, cela a été le standard pour la gestion de version (voir figure 1-2).

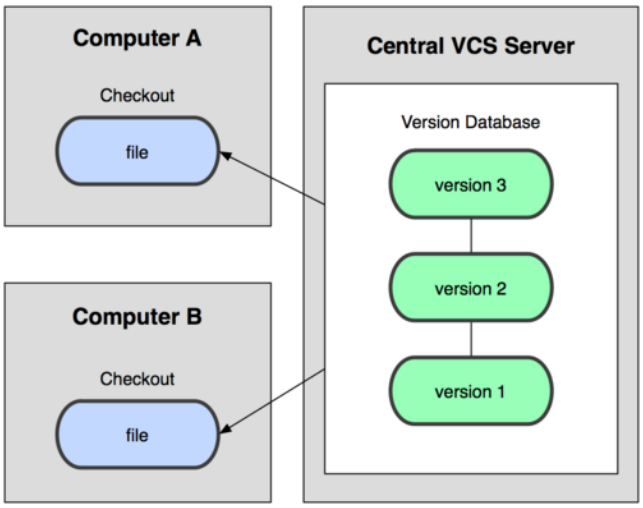

Figure 1.2: *Diagramme de la gestion de version centralisée.*

Ce schéma offre de nombreux avantages par rapport à la gestion de version locale. Par exemple,

chacun sait jusqu'à un certain point ce que tout les autres sont en train de faire sur le projet. Les administrateurs ont un contrôle fin des permissions et il est beaucoup plus facile d'administrer un CVCS que de gérer des bases de données locales.

Cependant ce système a aussi de nombreux défauts. Le plus visible est le point unique de panne que le serveur centralisé représente. Si ce serveur est en panne pendant une heure, alors durant cette heure, aucun client ne peut collaborer ou enregistrer les modifications issues de son travail. Si le disque dur du serveur central se corrompt, et s'il n'y a pas eu de sauvegarde, vous perdez absolument tout de l'historique d'un projet en dehors des sauvegardes locales que les gens auraient pu réaliser sur leur machines locales. Les systèmes de gestion de version locaux souffrent du même problème - dès qu'on a tout l'historique d'un projet sauvegardé à un endroit unique, on prend le risque de tout perdre.

1.1.3 Les systèmes de gestion de version distribués

C'est à ce moment que les systèmes de gestion de version distribués entrent en jeu (DVCSs en anglais pour *Distributed Version Control Systems*). Dans un DVCS (tel que Git, Mercurial, Bazaar or Darcs), les clients n'extraient plus seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît, et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données (voir figure 1-3).

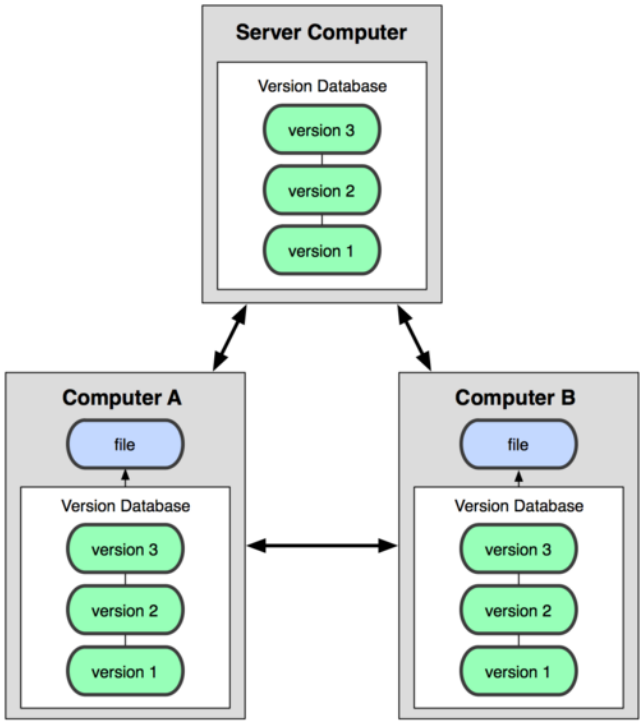

Figure 1.3: *Diagramme de gestion de version de contrôle centralisée.*

De plus, un grand nombre de ces systèmes gère particulièrement bien le fait d'avoir plusieurs dépôts avec lesquels travailler, vous permettant de collaborer avec différents groupes de personnes de manière différentes simultanément dans le même projet. Cela permet la mise en place de différentes chaînes de traitement qui ne sont pas réalisables avec les systèmes centralisés, tels que les modèles hiérarchiques.

1.2 Une rapide histoire de Git

Comme de nombreuses choses extraordinaires de la vie, Git est né avec une dose de destruction créative et de controverse houleuse. Le noyau Linux est un projet libre de grande envergure. Pour la plus grande partie de sa vie (1991-2002), les modifications étaient transmises sous forme de patchs et d'archives de fichiers. En 2002, le projet du noyau Linux commença à utiliser un DVCS propriétaire appelé BitKeeper.

En 2005, les relations entre la communauté développant le noyau linux et la société en charge du développement de BitKeeper furent rompues, et le statut de gratuité de l'outil fut révoqué. Cela poussa la communauté du développement de Linux (et plus particulièrement Linus Torvalds, le créateur de Linux) à développer leur propre outil en se basant sur les leçons apprises lors de l'utilisation de BitKeeper. Certains des objectifs du nouveau système étaient les suivants :

- Vitesse
- Conception simple
- Support pour les développements non linéaires (milliers de branches parallèles)
- Complètement distribué
- Capacité à gérer efficacement des projets d'envergure tels que le noyau Linux (vitesse et compacité des données)

Depuis sa naissance en 2005, Git a évolué et mûri pour être facile à utiliser tout en conservant ses qualités initiales. Il est incroyablement rapide, il est très efficace pour de grands projets et il a un incroyable système de branches pour des développements non linéaires (voir chapitre 3).

1.3 Rudiments de Git

Donc, qu'est-ce que Git en quelques mots ? Il est important de bien comprendre cette section, parce que si on comprend ce que Git est et les principes sur lesquels il repose, alors utiliser efficacement Git devient simple. Au cours de l'apprentissage de Git, essayez de libérer votre esprit de ce que vous pourriez connaître d'autres VCS, tels que Subversion et Perforce ; ce faisant, vous vous éviterez de petites confusions à l'utilisation de cet outil. Git enregistre et gère l'information très différemment des autres systèmes, même si l'interface utilisateur paraît similaire ; comprendre ces différences vous évitera des confusions à l'utilisation.

1.3.1 Des instantanés, pas des différences

La différence majeure entre Git et les autres VCS (Subversion et autres) réside dans la manière dont Git considère les données. Au niveau conceptuel, la plupart des autres VCS gèrent l'information comme une liste de modifications de fichiers. Ces systèmes (CVS, Subversion, Perforce, Bazaar et autres) considèrent l'information qu'il gèrent comme une liste de fichiers et les modifications effectuées sur chaque fichier dans le temps, comme illustré en figure 1-4.

Git ne gère pas et ne stocke pas les informations de cette manière. À la place, Git pense ses données plus comme un instantané d'un mini système de fichiers. A chaque fois que vous commitez ou enregistrez l'état du projet dans Git, il prend effectivement un instantané du contenu de votre espace de travail à ce moment et enregistre une référence à cet instantané. Pour être efficace, si les

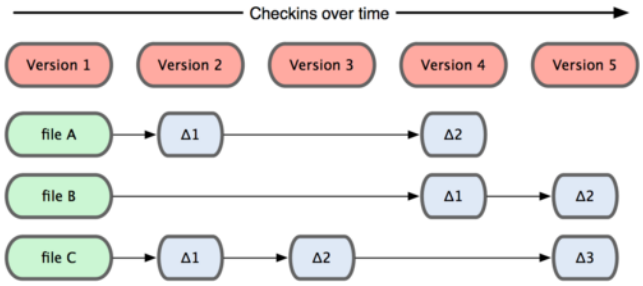

Figure 1.4: *D'autres systèmes sauvent l'information comme des modifications sur des fichiers.*

fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qui n'a pas été modifié. Git pense ses données plus à la manière de la figure 1-5.

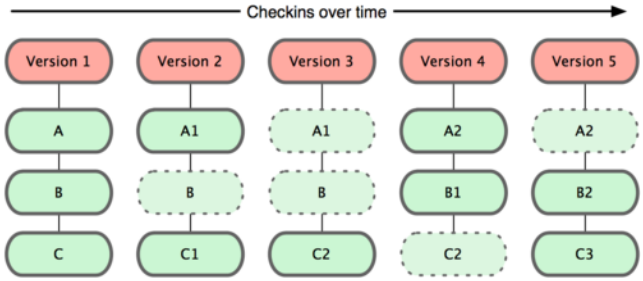

Figure 1.5: *Git stocke les données comme des instantanés du projet au cours du temps*

C'est une distinction importante entre Git et quasiment tous les autres VCSs. Git a reconsidéré quasiment tous les aspects de la gestion de version que la plupart des autres système ont copié des générations précédentes. Cela fait quasiment de Git un mini système de fichiers avec des outils incroyablement puissants construits au-dessus, plutôt qu'un simple VCS. Nous explorerons les bénéfices qu'il y a à penser les données de cette manière quand nous aborderons la gestion de branches au chapitre 3.

1.3.2 Presque toutes les opérations sont locales

La plupart des opérations de Git ne nécessite que des fichiers et ressources locales - généralement aucune information venant d'un autre ordinateur du réseau n'est nécessaire. Si vous êtes habitué à un CVCS où toutes les opérations sont ralenties par la latence des échanges réseau, cet aspect de Git vous fera penser que les dieux de la vitesse ont octroyé leurs pouvoirs à Git. Comme vous disposez de l'historique complet du projet localement sur votre disque dur, la plupart des opérations semblent instantanées.

Par exemple, pour parcourir l'historique d'un projet, Git n'a pas besoin d'aller le chercher sur un serveur pour vous l'afficher ; il n'a qu'à simplement le lire directement dans votre base de donnée locale. Cela signifie que vous avez quasi-instantanément accès à l'historique du projet. Si vous souhaitez connaître les modifications introduites entre la version actuelle d'un fichier et son état un mois auparavant, Git peut rechercher l'état du fichier un mois auparavant et réaliser le calcul de différence, au lieu d'avoir à demander cette différence à un serveur ou à devoir récupérer l'ancienne version sur le serveur pour calculer la différence localement.

Cela signifie aussi qu'il y a très peu de choses que vous ne puissiez réaliser si vous n'êtes pas connecté ou hors VPN. Si vous voyagez en train ou en avion et voulez avancer votre travail, vous pouvez continuer à gérer vos versions sans soucis en attendant de pouvoir de nouveau vous

connecter pour partager votre travail. Si vous êtes chez vous et ne pouvez avoir une liaison VPN avec votre entreprise, vous pouvez tout de même travailler. Pour de nombreux autres systèmes, faire de même est impossible ou au mieux très contraignant. Avec Perforce par exemple, vous ne pouvez pas faire grand'chose tant que vous n'êtes pas connecté au serveur. Avec Subversion ou CVS, vous pouvez éditer les fichiers, mais vous ne pourrez pas soumettre des modifications à votre base de données (car celle-ci est sur le serveur non accessible). Cela peut sembler peu important à priori, mais vous seriez étonné de découvrir quelle grande différence cela peut constituer à l'usage.

1.3.3 Git gère l'intégrité

Dans Git, tout est vérifié par une somme de contrôle avant d'être stocké et par la suite cette somme de contrôle, signature unique, sert de référence. Cela signifie qu'il est impossible de modifier le contenu d'un fichier ou d'un répertoire sans que Git ne s'en aperçoive. Cette fonctionnalité est ancrée dans les fondations de Git et fait partie intégrante de sa philosophie. Vous ne pouvez pas perdre des données en cours de transfert ou corrompre un fichier sans que Git ne puisse le détecter.

Le mécanisme que Git utilise pour réaliser les sommes de contrôle est appelé une empreinte SHA-1. C'est une chaîne de caractères composée de 40 caractères hexadécimaux (de '0' à '9' et de 'a' à 'f') calculée en fonction du contenu du fichier ou de la structure du répertoire considéré. Une empreinte SHA-1 ressemble à ceci :

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Vous trouverez ces valeurs à peu près partout dans Git car il les utilise pour tout. En fait, Git stocke tout non pas avec des noms de fichier, mais dans la base de données Git indexée par ces valeurs.

1.3.4 Généralement, Git ne fait qu'ajouter des données

Quand vous réalisez des actions dans Git, la quasi-totalité d'entre elles ne font qu'ajouter des données dans la base de données de Git. Il est très difficile de faire réaliser au système des actions qui ne soient pas réversibles ou de lui faire effacer des données d'une quelconque manière. Par contre, comme dans la plupart des systèmes de gestion de version, vous pouvez perdre ou corrompre des modifications qui n'ont pas encore été entrées en base ; mais dès que vous avez commité un instantané dans Git, il est très difficile de le perdre, spécialement si en plus vous synchronisez votre base de données locale avec un dépôt distant.

Cela fait de l'usage de Git un vrai plaisir, car on peut expérimenter sans danger de casser définitivement son projet. Pour une information plus approfondie sur la manière dont Git stocke ses données et comment récupérer des données qui pourraient sembler perdues, référez-vous au chapitre 9 « Sous le capot ».

1.3.5 Les trois états

Ici, il faut être attentif. Il est primordial de se souvenir de ce qui suit si vous souhaitez que le reste de votre apprentissage s'effectue sans difficulté. Git gère trois états dans lequel les fichiers peuvent résider : commité, modifié et indexé. Commité signifie que les données sont stockées en sécurité dans votre base de données locale. Modifié signifie que vous avez modifié le fichier mais

qu'il n'a pas encore été commité en base. Indexé signifie que vous avez marqué un fichier modifié dans sa version actuelle pour qu'il fasse partie du prochain instantané du projet.

Ceci nous mène aux trois sections principales d'un projet Git : le répertoire Git, le répertoire de travail et la zone d'index.

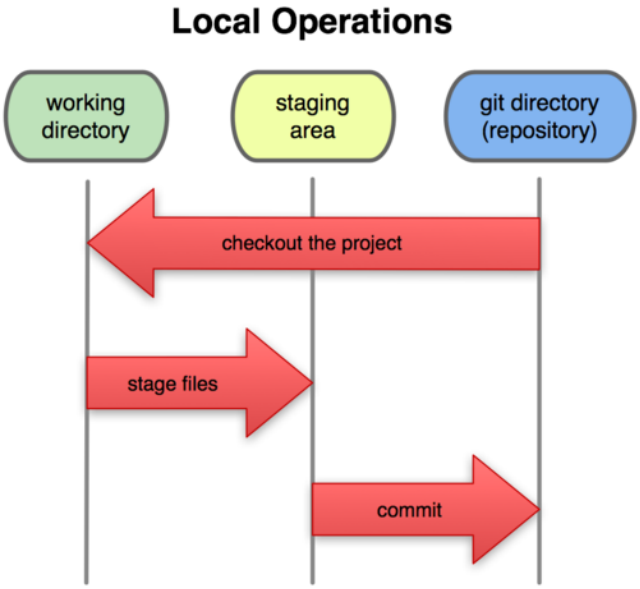

Figure 1.6: *Répertoire de travail, zone d'index et répertoire Git.*

Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Le répertoire de travail est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index est un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané.

L'utilisation standard de Git se passe comme suit :

1. Vous modifiez des fichiers dans votre répertoire de travail
2. Vous indexez les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index
3. Vous réalisez un commit, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans la base de donnée du répertoire Git.

Si une version particulière d'un fichier est dans le répertoire Git, il est considéré comme commité. S'il est modifié mais a été ajouté dans la zone d'index, il est indexé. S'il a été modifié depuis le dernier instantané mais n'a pas été indexé, il est modifié. Dans le chapitre 2, vous en apprendrez plus sur ces états et comment vous pouvez en tirer parti ou complètement les occulter.

1.4 Installation de Git

Commençons donc à utiliser Git. La première chose à faire est de l'installer. Vous pouvez l'obtenir par de nombreuses manières ; les deux principales sont de l'installer à partir des sources

ou d'installer un paquet existant sur votre plate-forme.

1.4.1 Installation depuis les sources

Si vous le pouvez, il est généralement conseillé d'installer Git à partir des sources, car vous obtiendrez la version la plus récente. Chaque nouvelle version de Git tend à inclure des améliorations utiles de l'interface utilisateur, donc récupérer la toute dernière version est souvent la meilleure option si vous savez compiler des logiciels à partir des sources. Comme la plupart du temps les distributions contiennent des version très anciennes de logiciels, à moins que vous ne travailliez sur une distribution très récente ou que vous n'utilisiez des backports, une installation à partir des sources peut être le meilleur choix.

Pour installer Git, vous avez besoin des bibliothèques suivantes : curl, zlib, openssl, expat, libiconv. Par exemple, si vous avez un système d'exploitation qui utilise yum (tel que Fedora) ou apt-get (tel qu'un système basé sur Debian), vous pouvez utiliser l'une des commandes suivantes pour installer les dépendances :

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Quand vous avez toutes les dépendances nécessaires, vous pouvez poursuivre et télécharger la dernière version de Git depuis le site :

http://git-scm.com/download

Puis, compiler et installer :

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Après ceci, vous pouvez obtenir Git par Git lui-même pour les mises à jour :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 Installation sur Linux

Si vous souhaitez installer Git sur Linux via un installateur d'application, vous pouvez généralement le faire via le système de gestion de paquet de base fourni avec votre distribution. Si vous êtes sur Fedora, vous pouvez utiliser yum :

```
$ yum install git-core
```

Si vous êtes sur un système basé sur Debian, tel qu'Ubuntu, essayez apt-get :

```
$ apt-get install git-core
```

1.4.3 Installation sur Mac

Il y a deux moyens simples d'installer Git sur Mac. Le plus simple et d'utiliser l'installateur graphique de Git que vous pouvez télécharger depuis les pages Google Code (voir figure 1-7) :

http://code.google.com/p/git-osx-installer

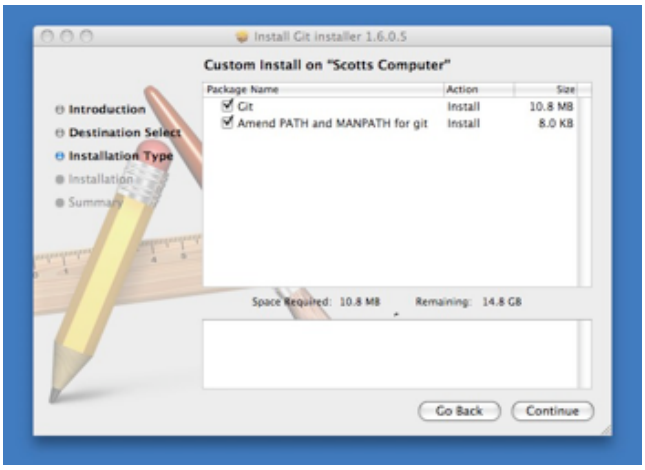

Figure 1.7: *Installateur OS X de Git.*

L'autre méthode consiste à installer Git par les MacPorts (http://www.macports.org). Si vous avez installé MacPorts, installez Git par :

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Vous n'avez pas à ajouter tous les extras, mais vous souhaiterez sûrement inclure +svn si vous êtes amené à utiliser Git avec des dépôts Subversion (voir chapitre 8).

1.4.4 Installation sur Windows

Installer Git sur Windows est très facile. Le projet msysGit fournit une des procédures d'installation les plus simples. Téléchargez simplement le fichier exe d'installateur depuis la page Google Code, et lancez-le :

http://code.google.com/p/msysgit

Après son installation, vous avez à la fois la version en ligne de commande (avec un client SSH utile pour la suite) ou l'interface graphique standard.

1.5 Paramétrage à la première utilisation de Git

Maintenant que vous avez installé Git sur votre système, vous voudrez personnaliser votre environnement Git. Vous ne devriez avoir à réaliser ces réglages qu'une seule fois ; ils persisteront lors

des mises à jour. Vous pouvez aussi les changer à tout instant en relançant les mêmes commandes.

Git contient un outil appelé git config pour vous permettre de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git. Ces variables peuvent être stockées dans trois endroits différents :

- Fichier `/etc/gitconfig` : Contient les valeurs pour tous les utilisateurs et tous les dépôts du système. Si vous passez l'option `--system` à `git config`, il lit et écrit ce fichier spécifiquement.
- Fichier `~/.gitconfig` : Spécifique à votre utilisateur. Vous pouvez forcer Git à lire et écrire ce fichier en passant l'option `--global`.
- Fichier `config` dans le répertoire Git (c'est à dire `.git/config`) du dépôt en cours d'utilisation : spécifique au seul dépôt en cours. Chaque niveau surcharge le niveau précédent, donc les valeurs dans `.git/config` surchargent celles de `/etc/gitconfig`.

Sur les systèmes Windows, Git recherche le fichier `.gitconfig` dans le répertoire `$HOME` (`C:\Documents and Settings\$USER` la plupart du temps). Il recherche tout de même `/etc/gitconfig`, bien qu'il soit relatif à la racine MSys, qui se trouve où vous aurez décidé d'installer Git sur votre système Windows.

1.5.1 Votre identité

La première chose à faire après l'installation de Git est de renseigner votre nom et votre adresse e-mail. C'est important car tous les commits Git utilisent cette information et elle est indélébile dans tous les commits que vous pourrez manipuler :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Encore une fois, cette étape n'est nécessaire qu'une fois si vous passez l'option `--global`, parce que Git utilisera toujours cette information pour tout ce que votre utilisateur fera sur ce système. Si vous souhaitez surcharger ces valeurs avec un nom ou une adresse e-mail différents pour un projet spécifique, vous pouvez lancer ces commandes sans option `--global` lorsque vous êtes dans ce projet.

1.5.2 Votre éditeur de texte

À présent que votre identité est renseignée, vous pouvez configurer l'éditeur de texte qui sera utilisé quand Git vous demande de saisir un message. Par défaut, Git utilise l'éditeur configuré au niveau système, qui est généralement Vi ou Vim. Si vous souhaitez utiliser un éditeur de texte différent, comme Emacs, vous pouvez entrer ce qui suit :

```
$ git config --global core.editor emacs
```

1.5.3 Votre outil de différences

Une autre option utile est le paramétrage de l'outil de différences à utiliser pour la résolution des conflits de fusion. Supposons que vous souhaitiez utiliser vimdiff :

```
$ git config --global merge.tool vimdiff
```

Git accepte kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, et opendiff comme outils valides de fusion. Vous pouvez aussi paramétrer un outil personnalisé ; référez-vous au chapitre 7 pour plus d'information sur cette procédure.

1.5.4 Vérifier vos paramètres

Si vous souhaitez vérifier vos réglages, vous pouvez utiliser la commande `git config --list` pour lister tous les réglages que Git a pu trouver jusqu'ici :

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Vous pourrez voir certains paramètres apparaître plusieurs fois, car Git lit les mêmes paramètres depuis plusieurs fichiers (`/etc/gitconfig` et `~/.gitconfig`, par exemple). Git utilise la dernière valeur pour chaque paramètre.

Vous pouvez aussi vérifier la valeur effective d'un paramètre particulier en tapant `git config <paramètre>` :

```
$ git config user.name
Scott Chacon
```

1.6 Obtenir de l'aide

Si vous avez besoin d'aide pour utiliser Git, il y a trois moyens d'obtenir les pages de manuel pour toutes les commandes de Git :

```
$ git help <verbe>
$ git <verbe> --help
$ man git-<verbe>
```

Par exemple, vous pouvez obtenir la page de manuel pour la commande config en lançant :

```
$ git help config
```

Ces commandes sont vraiment sympathiques car vous pouvez y accéder depuis partout, y compris hors connexion. Si les pages de manuel et ce livre ne sont pas suffisants, vous pouvez essayer les canaux `#git` ou `#github` sur le serveur IRC Freenode (irc.freenode.net). Ces canaux sont régulièrement peuplés de centaines de personnes qui ont une bonne connaissance de Git et sont souvent prêtes à aider.

1.7 Résumé

Vous devriez avoir à présent une compréhension initiale de ce que Git est et en quoi il est différent des CVCS que vous pourriez déjà avoir utilisés. Vous devriez aussi avoir une version de Git en état de fonctionnement sur votre système, paramétrée avec votre identité. Il est temps d'apprendre les bases d'utilisation de Git.

Chapitre 2

Les bases de Git

Si vous ne deviez lire qu'un chapitre avant de commencer à utiliser Git, c'est celui-ci. Ce chapitre couvre les commandes de base nécessaires pour réaliser la vaste majorité des activités avec Git. À la fin de ce chapitre, vous devriez être capable de configurer et initialiser un dépôt, commencer et arrêter le suivi de version de fichiers, d'indexer et valider des modifications. Nous vous montrerons aussi comment paramétrer Git pour qu'il ignore certains fichiers ou patrons de fichiers, comment revenir sur les erreurs rapidement et facilement, comment parcourir l'historique de votre projet et voir les modifications entre deux validations, et comment pousser et tirer les modifications avec des dépôts distants.

2.1 Démarrer un dépôt Git

Vous pouvez principalement démarrer un dépôt Git de deux manières. La première consiste à prendre un projet ou un répertoire existant et à l'importer dans Git. La seconde consiste à cloner un dépôt Git existant sur un autre serveur.

2.1.1 Initialisation d'un dépôt Git dans un répertoire existant

Si vous commencer à suivre un projet existant dans Git, vous n'avez qu'à vous positionner dans le répertoire du projet et saisir

```
$ git init
```

Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaire au dépôt — un squelette de dépôt Git. Pour l'instant, rien n'est encore suivi en version. (Cf. chapitre 9 pour plus d'information sur les fichiers contenus dans le répertoire `.git` que vous venez de créer.)

Si vous souhaitez commencer à suivre en version des fichiers existants (contrairement à un répertoire vide), vous devriez probablement commencer par indexer ces fichiers et faire une validation initiale. Vous pouvez réaliser ceci avec une poignée de commandes Git qui spécifient les fichiers que vous souhaitez suivre, suivi d'un commit :

```
$ git add *.c
$ git add README
$ git commit -m 'version initiale du projet'
```

Nous allons passer en revue ce que ces commandes font dans une petite minute. Pour l'instant, vous avez un dépôt git avec des fichiers sous gestion de version et un commit initial.

2.1.2 Cloner un dépôt existant

Si vous souhaitez obtenir une copie d'un dépôt Git existant — par exemple, un projet auquel vous aimeriez contribuer — la commande dont vous avez besoin s'appelle `git clone`. Si vous êtes familier avec d'autres systèmes de gestion de version tels que Subversion, vous noterez que la commande est 'clone' et non 'checkout'. C'est une distinction importante — Git reçoit une copie de quasiment toutes les données dont le serveur dispose. Toutes les versions de tous les fichiers pour l'historique du projet sont téléchargées quand vous lancez `git clone`. En fait, si le disque du serveur se corrompt, vous pouvez utiliser n'importe quel clone pour remettre le serveur dans l'état où il était au moment du clonage (vous pourriez perdre quelques paramètres du serveur, mais toutes les données sous gestion de version serait récupérées — Cf. chapitre 4 pour de plus amples détails).

Vous clonez un dépôt avec `git clone [url]`. Par exemple, si vous voulez cloner la bibliothèque Git Ruby appelée Grit, vous pouvez le faire de manière suivante :

```
$ git clone git://github.com/schacon/grit.git
```

Ceci crée un répertoire nommé « grit », initialise un répertoire `.git` à l'intérieur, récupère toutes les données de ce dépôt, et extrait une copie de travail de la dernière version. Si vous examinez le nouveau répertoire `grit`, vous y verrez les fichiers du projet, prêt à être modifiés ou utilisés. Si vous souhaitez cloner le dépôt dans un répertoire nommé différemment, vous pouvez spécifier le nom dans une option supplémentaire de la ligne de commande :

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Cette commande réalise la même chose que la précédente, mais le répertoire cible s'appelle mygrit.

Git dispose de différents protocoles de transfert que vous pouvez utiliser. L'exemple précédent utilise le protocole `git://`, mais vous pouvez aussi voir `http(s)://` ou `utilisateur@serveur:/chemin.git`, qui utilise le protocole de transfert SSH. Le chapitre 4 introduit toutes les options disponibles pour mettre en place un serveur Git, ainsi que leurs avantages et inconvénients.

2.2 Enregistrer des modifications dans le dépôt

Vous avez à présent un dépôt Git valide et une extraction ou copie de travail du projet. Vous devez faire quelques modifications et valider des instantanés de ces modifications dans votre dépôt chaque fois que votre projet atteint un état que vous souhaitez enregistrer.

Souvenez-vous que chaque fichier de votre copie de travail peut avoir deux états : sous suivi de version ou non suivi. Les fichiers suivis sont les fichiers qui appartenait déjà au dernier instantané ; ils peuvent être inchangés, modifiés ou indexés. Tous les autres fichiers sont non suivis — tout fichier de votre copie de travail qui n'appartenait pas à votre dernier instantané et n'a pas été indexé. Quand vous clonez un dépôt pour la première fois, tous les fichiers seront sous suivi de version et inchangés car vous venez tout juste de les enregistrer sans les avoir encore édités.

Au fur et à mesure que vous éditez des fichiers, Git les considère comme modifiés, car vous les avez modifiés depuis le dernier instantané. Vous indexez ces fichiers modifiés et vous enregistrez toutes les modifications indexées, puis ce cycle se répète. Ce cycle de vie est illustré par la figure 2-1.

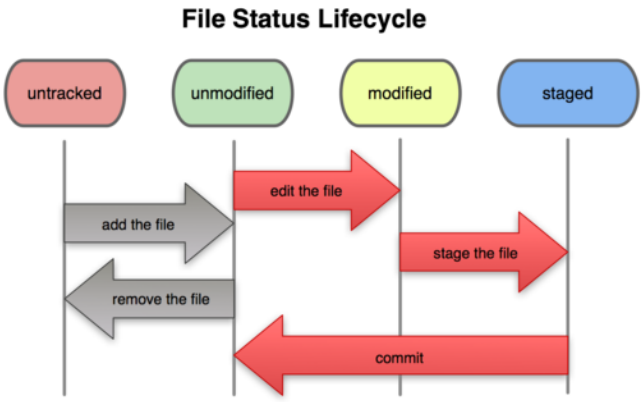

Figure 2.1: *Le cycle de vie des états des fichiers.*

2.2.1 Vérifier l'état des fichiers

L'outil principal pour déterminer quels fichiers sont dans quel état est la commande `git status`. Si vous lancez cette commande juste après un clonage, vous devriez voir ce qui suit :

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Ce message signifie que votre copie de travail est propre, en d'autres mots, aucun fichier suivi n'a été modifié. Git ne voit pas non plus de fichiers non-suivis, sinon ils seraient listés ici. Enfin, la commande vous indique sur quelle branche vous êtes. Pour l'instant, c'est toujours master, qui correspond à la valeur par défaut ; nous ne nous en soucierons pas maintenant. Dans le chapitre suivant, nous parlerons plus en détail des branches et des références.

Supposons que vous ajoutiez un nouveau fichier à votre projet, un simple fichier LISEZMOI. Si ce fichier n'existait pas auparavant, et que vous lancez la commande `git status`, vous verrez votre fichier non suivi comme ceci :

```
$ vim LISEZMOI
$ git status
# On branch master
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# LISEZMOI
nothing added to commit but untracked files present (use "git add" to track)
```

Vous pouvez constater que votre nouveau fichier LISEZMOI n'est pas en suivi de version, car il apparaît dans la section « Untracked files » de l'état de la copie de travail. « Untracked » signifie simplement que Git détecte un fichier qui n'était pas présent dans le dernier instantané ; Git ne le placera sous suivi en version que quand vous lui indiquerez de le faire. Ce comportement permet de ne pas placer accidentellement sous suivi de version des fichiers binaires générés ou d'autres fichiers que vous ne voulez pas inclure. Mais vous voulez inclure le fichier LISEZMOI dans l'instantané, alors commençons à suivre ce fichier.

2.2.2 Placer de nouveaux fichiers sous suivi de version

Pour commencer à suivre un nouveau fichier, vous utilisez la commande `git add`. Pour commencer à suivre le fichier LISEZMOI, vous pouvez entrer ceci :

```
$ git add LISEZMOI
```

Si vous lancez à nouveau le commande status, vous pouvez constater que votre fichier LISEZMOI est maintenant suivi et indexé :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
#
```

Vous pouvez affirmer qu'il est indexé car il apparaît dans la section « Changes to be committed » (Modifications à enregistrer).Si vous enregistrez à ce moment, la version du fichier à l'instant où vous lancez `git add` est celle qui appartiendra à l'instantané. Vous pouvez vous souvenir que lorsque vous avez précédemment lancé `git init`, vous avez ensuite lancé `git add (fichiers)` — c'était bien sur pour commencer à placer sous suivi de version les fichiers de votre répertoire de travail. La commande git add accepte en paramètre un chemin qui correspond à un fichier ou un répertoire ; dans le cas d'un répertoire, la commande ajoute récursivement tous les fichiers de ce répertoire.

2.2.3 Indexer des fichiers modifiés

Maintenant, modifions un fichiers qui est déjà sous suivi de version. Si vous modifiez le fichier sous suivi de version appelé `benchmarks.rb` et que vous lancez à nouveau votre commande `status`, vous verrez ceci :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

Le fichier benchmarks.rb apparaît sous la section nommée « Changed but not updated » ce qui signifie que le fichier sous suivi de version a été modifié dans la copie de travail mais n'est pas encore indexé. Pour l'indexer, il faut lancer la commande `git add` (qui est une commande multi-usage — elle peut être utilisée pour placer un fichier sous suivi de version, pour indexer un fichier ou pour d'autres actions telles que marquer comme résolu des conflits de fusion de fichiers). Lançons maintenant `git add` pour indexer le fichier benchmarks.rb, et relançons la commande `git status` :

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
# modified:   benchmarks.rb
#
```

A présent, les deux fichiers sont indexés et feront partie de la prochaine validation. Mais supposons que vous souhaitiez apporter encore une petite modification au fichier benchmarks.rb avant de réellement valider la nouvelle version. Vous l'ouvrez à nouveau, réalisez la petite modification et vous voilà prêt à valider. Néanmoins, vous lancez `git status` une dernière fois :

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
# modified:   benchmarks.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
```

```
# modified:   benchmarks.rb
#
```

Que s'est-il donc passé ? À présent, benchmarks.rb apparaît à la fois comme indexé et non indexé. En fait, Git indexe un fichier dans son état au moment où la commande `git add` est lancée. Si on valide les modifications maintenant, la version de benchmarks.rb qui fera partie de l'instantané est celle correspondant au moment où la commande `git add benchmarks.rb` a été lancée, et non la version actuellement présente dans la copie de travail au moment où la commande git commit est lancée. Si le fichier est modifié après un `git add`, il faut relancer `git add` pour prendre en compte l'état actuel dans la copie de travail :

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
# modified:   benchmarks.rb
#
```

2.2.4 Ignorer des fichiers

Il apparaît souvent qu'un type de fichiers présent dans la copie de travail ne doit pas être ajouté automatiquement ou même apparaître comme fichier potentiel pour le suivi de version. Ce sont par exemple des fichiers générés automatiquement tels que les fichiers de journaux ou de sauvegardes produits par l'outil que vous utilisez. Dans un tel cas, on peut énumérer les patrons de noms de fichiers à ignorer dans un fichier .gitignore. Voici ci-dessous un exemple de fichier .gitignore :

```
$ cat .gitignore
*.[oa]
*~
```

La première ligne ordonne à Git d'ignorer tout fichier se terminant en .o ou .a — des fichiers objet ou archive qui sont généralement produits par la compilation d'un programme. La seconde ligne indique à Git d'ignorer tous les fichiers se terminant par un tilde (~), ce qui est le cas des noms des fichiers temporaires pour de nombreux éditeurs de texte tels qu'Emacs. On peut aussi inclure un répertoire log, tmp ou pid, ou le répertoire de documentation générée automatiquement, ou tout autre fichier. Renseigner un fichier .gitignore avant de commencer à travailler est généralement une bonne idée qui évitera de valider par inadvertance des fichiers qui ne doivent pas apparaître dans le dépôt Git.

Les règles de construction des patrons à placer dans le fichier .gitignore sont les suivantes :

- Les lignes vides ou commençant par # sont ignorée
- Les patrons standards de fichiers sont utilisables

- Si le patron se termine par un slash (`/`), le patron dénote un répertoire
- Un patron commençant par un point d'exclamation (`!`) est inversé.

Les patrons standards de fichiers sont des expressions régulières simplifiées utilisées par les shells. Un astérisque (`*`) correspond à un ou plusieurs caractères ; `[abc]` correspond à un des trois caractères listés dans les crochets, donc a ou b ou c ; un point d'interrogation (`?`) correspond à un unique caractère ; des crochets entourant des caractères séparés par un signe moins (`[0-9]`) correspond à un caractère dans l'intervalle des deux caractères indiqués, donc ici de `0` à `9`.

Voici un autre exemple de fichier .gitignore :

```
# un commentaire, cette ligne est ignorée
*.a       # pas de fichier .a
!lib.a    # mais suivre en version lib.a malgré la règle précédente
/TODO     # ignorer uniquement le fichier TODO à la racine du projet
build/    # ignorer tous le fichiers dans le répertoire build
doc/*.txt # ignorer doc/notes.txt, mais pas doc/server/arch.txt
```

2.2.5 Inspecter les modifications indexées et non indexées

Si le résultat de la commande `git status` est encore trop vague — lorsqu'on désire savoir non seulement quels fichiers ont changé mais aussi ce qui a changé dans ces fichiers — on peut utiliser la commande `git diff`. Cette commande sera traitée en détail plus loin ; mais elle sera vraisemblablement utilisée le plus souvent pour répondre aux questions suivantes : qu'est-ce qui a été modifié mais pas encore indexé ? Quelle modifications a été indexée et est prête pour la validation ? Là où `git status` répond de manière générale à ces questions, `git diff` montre les lignes exactes qui ont été ajoutées, modifiées ou effacées — le patch en somme.

Supposons que vous éditez et indexez le fichier LISEZMOI et que vous éditez le fichier benchmarks.rb sans l'indexer. Si vous lancez la commande `status`, vous verrez ceci :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   LISEZMOI
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

Pour visualiser ce qui a été modifié mais pas encore indexé, tapez `git diff` sans autre argument :

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

Cette commande compare le contenu du répertoire de travail avec la zone d'index. Le résultat vous indique les modifications réalisées mais non indexées.

Si vous souhaitez visualiser les modifications indexées qui feront partie de la prochaine validation, vous pouvez utiliser `git diff --cached` (avec les versions 1.6.1 et supérieures de Git, vous pouvez aussi utiliser `git diff --staged`, qui est plus mnémotechnique). Cette commande compare les fichiers indexés et le dernier instantané :

```
$ git diff --cached
diff --git a/LISEZMOI b/LISEZMOI
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/LISEZMOI2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

Il est important de noter que `git diff` ne montre pas les modifications réalisées depuis la dernière validation — seulement les modifications qui sont non indexées. Cela peut introduire une confusion car si tous les fichiers modifiés ont été indexés, `git diff` n'indiquera aucun changement.

Par exemple, si vous indexez le fichier benchmarks.rb et l'éditez en suite, vous pouvez utiliser `git diff` pour visualiser les modifications indexées et non indexées de ce fichier :

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
```

```
#
# Changes to be committed:
#
# modified:   benchmarks.rb
#
# Changed but not updated:
#
# modified:   benchmarks.rb
#
```

A présent, vous pouvez utiliser `git diff` pour visualiser les modifications non indexées :

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

et `git diff --cached` pour visualiser ce qui a été indexé jusqu'à maintenant :

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
          @commit.parents[0].parents[0].parents[0]
        end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
        run_code(x, 'commits 2') do
          log = git.commits('master', 15)
          log.size
```

2.2.6 Valider vos modifications

Votre zone d'index est dans l'état désiré, vous pouvez valider vos modifications. Souvenez-vous que tout ce qui encore non indexé — tous les fichiers qui ont été créés ou modifiés mais n'ont pas subi de `git add` depuis ne feront pas partie de la prochaine validation. Ils resteront en tant que fichiers modifiés sur votre disque.

Dans notre cas, la dernière fois que vous avez lancé `git status`, vous avez vérifié que tout était indexé, et vous êtes donc prêt à valider vos modifications. La manière la plus simple de valider est de taper `git commit` :

```
$ git commit
```

Cette action lance votre éditeur par défaut (qui est paramétré par la variable d'environnement `$EDITOR` de votre shell — habituellement vim ou Emacs, mais vous pouvez le paramétrer spécifiquement pour git en utilisant la commande `git config --global core.editor` comme nous l'avons vu au chapitre 1).

L'éditeur affiche le texte suivant :

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   LISEZMOI
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Vous constatez que le message de validation par défaut contient une ligne vide suivie en commentaire le résultat de la commande `git status`. Vous pouvez effacer ces lignes de commentaire et saisir votre propre message de validation, ou vous pouvez les laisser en place vous aider à vous rappeler de ce que vous êtes en train de valider (pour un rappel plus explicite de ce que vous avez modifié, vous pouvez aussi passer l'option `-v` à la commande `git commit`. Cette option place le résultat du diff en commentaire dans l'éditeur pour vous permettre de visualiser exactement ce que vous avez modifié). Quand vous quittez l'éditeur (après avoir sauvegardé le message), Git crée votre commit avec ce message de validation (après avoir retiré les commentaires et le diff).

D'une autre manière, vous pouvez spécifier votre message de validation en ligne avec la commande `commit` en le saisissant après l'option `-m`, de cette manière :

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 LISEZMOI
```

A présent, vous avez créé votre premier commit ! Vous pouvez constater que le commit vous fournit quelques information sur lui-même : sur quelle branche vous avez validé (master), quelle est sa somme de contrôle SHA-1 (`463dc4f`), combien de fichiers ont été modifiés, et quelques statistiques sur les lignes ajoutées et effacées dans ce commit.

Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. Tout ce que vous n'avez pas indexé est toujours en état modifié ; vous pouvez réaliser une nouvelle validation pour l'ajouter à l'historique. A chaque validation, vous enregistrez un instantané du projet en forme de jalon auquel vous pourrez revenir ou comparer votre travail ultérieur.

2.2.7 Éliminer la phase d'indexation

Bien qu'il soit incroyablement utile de pouvoir organiser les commits exactement comme on l'entend, la gestion de la zone d'index est parfois plus complexe que nécessaire dans une utilisation normale. Si vous souhaitez éviter la phase de placement des fichiers dans la zone d'index, Git fournit un raccourcis très simple. L'ajout de l'option `-a` à la commande `git commit` ordonne à Git de placer automatiquement tout fichier déjà en suivi de version dans la zone d'index avant de réaliser la validation, évitant ainsi d'avoir à taper les commandes `git add` :

```
$ git status
# On branch master
#
# Changed but not updated:
#
# modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Notez bien que vous n'avez pas eu à lancer `git add` sur le fichier benchmarks.rb avant de valider.

2.2.8 Effacer des fichiers

Pour effacer un fichier de Git, vous devez l'éliminer des fichiers en suivi de version (plus précisément, l'effacer dans la zone d'index) puis valider. La commande `git rm` réalise cette action mais efface aussi ce fichier de votre copie de travail de telle sorte que vous ne le verrez pas réapparaître comme fichier non suivi en version à la prochaine validation.

Si vous effacez simplement le fichier dans votre copie de travail, il apparaît sous la section "Changed but not updated" (c'est-à-dire, *non indexé*) dans le résultat de `git status` :

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Ensuite, si vous lancez `git rm`, l'effacement du fichier est indexé :

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

Lors de la prochaine validation, le fichier sera absent et non-suivi en version. Si vous avez auparavant modifié et indexé le fichier, son élimination doit être forcée avec l'option `-f`. C'est une mesure de sécurité pour empêcher un effacement accidentel de données qui n'ont pas encore été enregistrées dans un instantané et qui seraient définitivement perdues.

Un autre scénario serait de vouloir abandonner le suivi de version d'un fichier tout en le conservant dans la copie de travail. Ceci est particulièrement utile lorsqu'on a oublié de spécifier un patron dans le fichier `.gitignore` et on a accidentellement ajouté un fichier dans l'instantané, tel qu'un gros fichier de journal ou une série d'archives de compilation `.a`. Pour réaliser ce scénario, utilisez l'option `--cached` :

```
$ git rm --cached readme.txt
```

Vous pouvez spécifier des noms de fichiers ou de répertoires, ou des patrons de fichiers à la commande `git rm`. Cela signifie que vous pouvez lancer des commandes telles que

```
$ git rm log/\*.log
```

Notez bien l'antislash (`\`) devant `*`. Il est nécessaire d'échapper le caractère `*` car Git utilise sa propre expansion de nom de fichier en addition de l'expansion du shell. Cette commande efface tous les fichiers avec l'extension `.log` présents dans le répertoire `log/`. Vous pouvez aussi lancer une commande telle que :

```
$ git rm \*~
```

Cette commande élimine tous les fichiers se terminant par ~.

2.2.9 Déplacer des fichiers

À la différence des autres VCS, Git ne suit pas explicitement les mouvements des fichiers. Si vous renommez un fichier suivi par Git, aucune méta-donnée indiquant le renommage n'est stockée par Git. Néanmoins, Git est assez malin pour s'en apercevoir après coup — la détection de mouvement de fichier sera traitée plus loin.

De ce fait, que Git ait une commande `mv` peut paraître trompeur. Si vous souhaitez renommer un fichier dans Git, vous pouvez lancer quelque chose comme

```
$ git mv nom_origine nom_cible
```

et cela fonctionne. En fait, si vous lancez quelque chose comme ceci et inspectez le résultat d'une commande `status`, vous constaterez que Git gère le renommage de fichier :

```
$ git mv LISEZMOI.txt LISEZMOI
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    LISEZMOI.txt -> LISEZMOI
#
```

Néanmoins, cela revient à lancer les commandes suivantes :

```
$ mv LISEZMOI.txt LISEZMOI
$ git rm LISEZMOI.txt
$ git add LISEZMOI
```

Git trouve implicitement que c'est un renommage, donc cela importe peu si vous renommez un fichier de cette manière ou avec la commande `mv`. La seule différence réelle est que `mv` ne fait qu'une commande à taper au lieu de trois — c'est une commande de convenance. Le point principal est que vous pouvez utiliser n'importe quel outil pour renommer un fichier, et traiter les commandes `add`/`rm` plus tard, avant de valider la modification.

2.3 Visualiser l'historique des validations

Après avoir créé plusieurs commits ou si vous avez cloné un dépôt ayant un historique de commits, vous souhaitez probablement revoir le fil des évènements. La commande `git log` est l'outil le plus basique et puissant pour cet objet.

Les exemples qui suivent utilisent un projet très simple nommé simplegit utilisé pour les démonstrations. Pour récupérer le projet, lancez

```
git clone git://github.com/schacon/simplegit-progit.git
```

Lorsque vous lancez `git log` dans le répertoire de ce projet, vous devriez obtenir un résultat qui ressemble à ceci :

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Par défaut, `git log` invoqué sans argument énumère en ordre chronologique inversé les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier. Comme vous le remarquez, cette commande indique chaque commit avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.

`git log` dispose d'un très grand nombre d'options permettant de paramétrer exactement ce que l'on cherche à voir. Nous allons détailler quelques unes des plus utilisées.

Une des options les plus utiles est `-p`, qui montre les différences introduites entre chaque validation. Vous pouvez aussi utiliser `-2` qui limite la sortie de la commande aux deux entrées les plus récentes :

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code
```

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Cette option affiche la même information mais avec un diff suivant directement chaque entrée. C'est très utile pour des revues de code ou pour naviguer rapidement à travers l'historique des modifications qu'un collaborateur a apportées.

Vous pouvez aussi utiliser une liste d'options de résumé avec `git log`. Par exemple, si vous souhaitez visualiser des statistiques résumées pour chaque commit, vous pouvez utiliser l'option `--stat` :

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 LISEZMOI           |    6 ++++++
 Rakefile           |   23 +++++++++++++++++++++++
 lib/simplegit.rb   |   25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+), 0 deletions(-)
```

Comme vous pouvez le voir, l'option `--stat` affiche sous chaque entrée de validation une liste des fichiers modifiés, combien de fichiers ont été changés et combien de lignes ont été ajoutées ou retirées dans ces fichiers. Elle ajoute un résumé des informations en fin de sortie. Une autre option utile est `--pretty`. Cette option modifie le journal vers un format différent. Quelques options incluses sont disponibles. L'option `oneline` affiche chaque commit sur une seule ligne, ce qui peut s'avérer utile lors de la revue d'un long journal. De plus, les options `short`, `full` et `fuller` montrent le résultat à peu de choses près dans le même format mais avec de plus en plus d'information :

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

L'option la plus intéressante est `format` qui permet de décrire précisément le format de sortie. C'est spécialement utile pour générer des sorties dans un format facile à analyser par une machine — lorsqu'on spécifie intégralement et explicitement le format, on s'assure qu'il ne changera pas au gré des mises à jour de Git :

```
$ git log --pretty=format:"%h — %an, %ar : %s"
ca82a6d — Scott Chacon, 11 months ago : changed the version number
085bb3b — Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 — Scott Chacon, 11 months ago : first commit
```

Le tableau 2-1 liste les options de formatage les plus utiles.

```
Option Description du formatage
%H Somme de contrôle du commit
%h Somme de contrôle abrégée du commit
%T Somme de contrôle de l'arborescence
%t Somme de contrôle abrégée de l'arborescence
%P Sommes de contrôle des parents
%p Sommes de contrôle abrégées des parents
%an Nom de l'auteur
%ae e-mail de l'auteur
%ad Date de l'auteur (au format de l'option -date=)
%ar Date relative de l'auteur
%cn Nom du validateur
%ce e-mail du validateur
%cd Date du validateur
%cr Date relative du validateur
%s Sujet
```

Vous pourriez vous demander quelle est la différence entre *auteur* et *validateur*. L'auteur est la personne qui a réalisé initialement le travail, alors que le validateur est la personne qui a effectivement validé ce travail en gestion de version. Donc, si quelqu'un envoie patch à un projet et un des membres du projet l'applique, les deux personnes reçoivent le crédit — l'écrivain en tant qu'auteur,

et le membre du projet en tant que validateur. Nous traiterons plus avant de cette distinction au chapitre 5.

Les options oneline et format sont encore plus utiles avec une autre option `log` appelée `--graph`. Cette option ajoute un joli graphe en caractères ASCII pour décrire l'historique des branches et fusions, ce que nous pouvons visualiser pour notre copie du dépôt de Grit :

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Les options ci-dessus ne sont que des options simples de format de sortie de `git log` — il y en a de nombreuses autres. Le tableau 2-2 donne une liste des options que nous avons traitées ainsi que d'autres options communément utilisées accompagnées de la manière dont elles modifient le résultat de la commande log.

```
Option Description
-p Affiche le patch appliqué par chaque commit
--stat Affiche les statistiques de chaque fichier pour chaque commit
--shortstat N'affiche que les ligne modifiées/insérées/effacées de l'option --stat
--name-only Affiche la liste des fichiers modifiés après les informations du commit
--name-status Affiche la liste des fichiers affectés accompagnés des informations d'ajout/
modification/suppression
--abbrev-commit N'affiche que les premiers caractères de la somme de contrôle SHA-1
--relative-date Affiche la date en format relatif (par exemple "2 weeks ago" : il y a deux semaines) au lieu d
--graph Affiche en caractère ASCII le graphe de branches et fusions en vis-à-
vis de l'historique
--pretty=<format>Affiche les commits dans un format alternatif. Les formats incluent oneline, short, full, f
```

2.3.1 Limiter la longueur de l'historique

En complément des options de formatage de sortie, git log est pourvu de certaines options de limitation utiles — des options qui permettent de restreindre la liste à un sous-ensemble de commits. Vous avez déjà vu une de ces options — l'option `-2` qui ne montre que le deux derniers commits. En fait, on peut utiliser `-<n>`, ou `n` correspond au nombre de commits que l'on cherche à visualiser en partant des plus récents. En vérité, il est peu probable que vous utilisiez cette option, parce que Git injecte par défaut sa sortie dans un outil de pagination qui permet de la visualiser page à page.

Cependant, les options de limitation portant sur le temps, telles que `--since` (depuis) et `--until` (jusqu'à) sont très utiles. Par exemple, le commande suivante affiche la liste des commits des deux dernières semaines :

```
$ git log --since=2.weeks
```

Cette commande fonctionne avec de nombreux formats — vous pouvez indiquer une date spécifique (2008-01-05) ou une date relative au présent telle que « 2 years 1 day 3 minutes ago ».

Vous pouvez aussi restreindre la liste aux commits vérifiant certains critères de recherche. L'option `--author` permet de filtrer sur un auteur spécifique, et l'option `--grep` permet de chercher des mots clés dans les messages de validation. Notez que si vous cherchez seulement des commits correspondant simultanément aux deux critères, vous devez ajouter l'option `--all-match`, car par défaut ces commandes retournent les commits vérifiant au moins un critère lors de recherche de chaînes de caractères.

La dernière option vraiment utile à `git log` est la spécification d'un chemin. Si un répertoire ou un nom de fichier est spécifié, le journal est limité aux commits qui ont introduit des modifications aux fichiers concernés. C'est toujours la dernière option de la commande, souvent précédée de deux tirets (`--`) pour séparer le chemin des options précédentes.

Le tableau 2-3 récapitule les options que nous venons de voir ainsi que quelques autres pour référence.

```
Option Description
-(n) N'affiche que les n derniers commits
--since, --after Limite l'affichage aux commits réalisés après la date spécifiée
--until, --before Limite l'affichage aux commits réalisés avant la date spécifiée
--author Ne montre que les commits dont le champ auteur correspond à la chaîne passée en argument
--committer Ne montre que les commits dont le champ validateur correspond à la chaîne passée en argument
```

Par exemple, si vous souhaitez visualiser quels commits modifiant les fichiers de test dans l'historique du source de Git ont été validés par Junio Hamano et n'étaient pas des fusions durant le mois d'octobre 2008, vous pouvez lancer ce qui suit :

```
$ git log --pretty="%h — %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b — Fix testcase failure when extended attribute
acd3b9e — Enhance hold_lock_file_for_{update,append}()
f563754 — demonstrate breakage of detached checkout wi
d1a43f2 — reset --hard/read-tree --reset -u: remove un
51a94af — Fix "checkout --track -b newbranch" on detac
b0ad11e — pull: allow "git pull origin $something:$cur
```

A partir des 20 000 commits constituant l'historique des sources de Git, cette commande extrait les 6 qui correspondent aux critères.

2.3.2 Utiliser une interface graphique pour visualiser l'historique

Si vous préférez utiliser un outil plus graphique pour visualiser l'historique d'un projet, vous pourriez jeter un œil à un programme distribué avec Git nommé gitk. Gitk est un outil graphique

mimant les fonctionnalités de `git log`, et il donne accès à quasiment toutes les options de filtrage de `git log`. Si vous tapez gitk en ligne de commande, vous devriez voir une interface ressemblant à la figure 2-2.

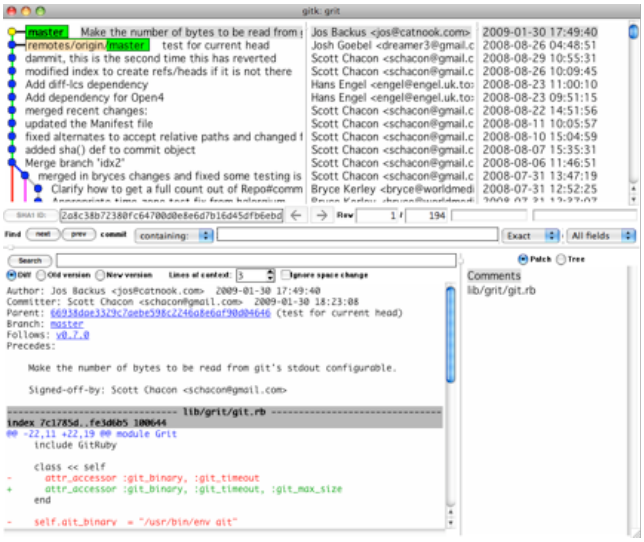

Figure 2.2: *Le visualiseur d'historique gitk*

Vous pouvez voir l'historique des commits dans la partie supérieure de la fenêtre avec un graphique d'enchaînement. Le visualisateur de diff dans la partie inférieure de la fenêtre affiche les modifications introduites par le commit sélectionné.

2.4 Annuler des actions

À tout moment, vous pouvez désirer annuler une de vos dernières actions. Dans cette section, nous allons passer en revue quelques outils de base permettant d'annuler des modifications. Il faut être très attentif car certaines de ces annulations sont définitives (elles ne peuvent pas être elle-même annulées). C'est donc un des rares cas d'utilisation de Git où des erreurs de manipulation peuvent entraîner des pertes définitives de données.

2.4.1 Modifier le dernier commit

Une des annulations les plus communes apparaît lorsqu'on valide une modification trop tôt en oubliant d'ajouter certains fichiers, ou si on se trompe dans le message de validation. Si vous souhaitez rectifier cette erreur, vous pouvez valider le complément de modification avec l'option `--amend` :

```
$ git commit --amend
```

Cette commande prend en compte la zone d'index et l'utilise pour le commit. Si aucune modification n'a été réalisée depuis la dernière validation (par exemple en lançant cette commande immédiatement après la dernière validation), alors l'instantané sera identique et la seule modification à introduire sera le message de validation.

L'éditeur de message de validation démarre, mais il contient déjà le message de la validation précédente. Vous pouvez éditer ce message normalement, mais il écrasera le message de la validation précédente.

Par exemple, si vous validez une version puis réalisez que vous avez oublié de spécifier les modifications d'un fichier, vous pouvez taper les commandes suivantes :

```
$ git commit -m 'validation initiale'
$ git add fichier_oublie
$ git commit --amend
```

Les trois dernières commandes donnent lieu à la création d'un unique commit — la seconde validation remplace le résultat de la première.

2.4.2 Désindexer un fichier déjà indexé

Les deux sections suivantes démontrent comment bricoler les modifications dans votre zone d'index et votre zone de travail. Un point sympathique est que la commande permettant de connaître l'état de ces deux zones vous rappelle aussi comment annuler les modifications. Par exemple, supposons que vous avez modifié deux fichiers et voulez les valider comme deux modifications indépendantes, mais que vous ayez tapé accidentellement `git add *` et donc indexé les deux. Comment annuler l'indexation d'un des fichiers ? La commande `git status` vous rappelle :

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   LISEZMOI.txt
#       modified:   benchmarks.rb
#
```

Juste sous le texte « Changes to be committed », elle vous indique d'utiliser `git reset HEAD <fichier>...` pour désindexer un fichier. Utilisons donc ce conseil pour désindexer le fichier benchmarks.rb :

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   LISEZMOI.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
```

```
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

La commande à taper peut sembler étrange mais elle fonctionne. Le fichier benchmark.rb est modifié mais de retour à l'état non indexé.

2.4.3 Réinitialiser un fichier modifié

Que faire si vous réalisez que vous ne souhaitez pas conserver les modifications au fichier benchmark.rb ? Comment le réinitialiser facilement, le ramener à l'état qu'il avait dans le dernier instantané (ou lors du clonage, ou dans l'état dans lequel vous l'avez obtenu dans votre copie de travail) ? Heureusement, `git status` est secourable. Dans le résultat de la dernière commande, la zone de travail ressemble à ceci :

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

Cela vous indique de façon explicite comment annuler des modifications que vous avez faites (du moins, les nouvelles versions de Git, 1.6.1 et supérieures le font, si vous avez une version plus ancienne, nous vous recommandons de la mettre à jour pour bénéficier de ces fonctionnalités pratiques). Faisons comme indiqué :

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   LISEZMOI
#
```

Vous pouvez constater que les modifications ont été annulées. Vous devriez aussi vous apercevoir que c'est une commande dangereuse : toutes les modifications que vous auriez réalisées sur ce fichier ont disparu — vous venez tout juste de l'écraser avec un autre fichier. N'utilisez jamais cette commande à moins d'être vraiment sûr de ne pas vouloir de ces modifications. Si vous souhaitez seulement écarter momentanément cette modification, nous verrons comment mettre de côté et créer des branches dans le chapitre suivant ; ce sont de meilleures façons de procéder. Souvenez-vous, tout ce qui a été validé dans Git peut quasiment toujours être récupéré. Y compris des commits sur des branches qui ont été effacées ou des commits qui ont été écrasés par une validation avec l'option `--amend` (se référer au chapitre 9 pour la récupération de données). Cependant, tout ce que vous perdez avant de l'avoir validé n'a aucune chance d'être récupérable via Git.

2.5 Travailler avec des dépôts distants

Pour pouvoir collaborer sur un projet Git, il est nécessaire de connaître comment gérer les dépôts distants. Les dépôts distants sont des versions de votre projet qui sont hébergées sur Internet ou le réseau. Vous pouvez en avoir plusieurs, pour lesquels vous pouvez avoir des droits soit en lecture seule, soit en lecture/écriture. Collaborer avec d'autres personnes consiste à gérer ces dépôts distants, en poussant ou tirant des données depuis et vers ces dépôts quand vous souhaitez partager votre travail.

Gérer des dépôts distants inclut savoir comment ajouter des dépôts distants, effacer des dépôts distants qui ne sont plus valides, gérer des branches distantes et les définir comme suivie ou non, et plus encore. Dans cette section, nous traiterons des commandes de gestion distante.

2.5.1 Afficher les dépôts distants

Pour visualiser les serveur distants que vous avez enregistrés, vous pouvez lancer le commande git remote. Elle liste les noms des différentes étiquettes distantes que vous avez spécifiées. Si vous avez cloné un dépôt, vous devriez au moins voir l'origine origin — c'est-à-dire le nom par défaut que Git donne au serveur à partir duquel vous avez cloné :

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Vous pouvez aussi spécifier `-v`, qui vous montre l'URL que Git a stocké pour nom court à étendre :

```
$ git remote -v
origin git://github.com/schacon/ticgit.git
```

Si vous avez plus d'un dépôt distant, la commande précédente les liste tous. Par exemple, mon dépôt Grit ressemble à ceci.

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

Cela signifie que nous pouvons tirer très facilement des contributions depuis certains utilisateurs. Mais il est à noter que seul le dépôt distant origin utilise une URL SSH, ce qui signifie que c'est le seul sur lequel je peux pousser (nous traiterons de ceci au chapitre 4).

2.5.2 Ajouter des dépôts distants

J'ai expliqué et donné des exemples d'ajout de dépôts distants dans les chapitres précédents, mais voici spécifiquement comment faire. Pour ajouter un nouveau dépôt distant Git comme nom court auquel il est facile de faire référence, lancez `git remote add [nomcourt] [url]` :

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

Maintenant, vous pouvez utiliser le mot-clé pb sur la ligne de commande au lieu de l'URL complète. Par exemple, si vous voulez récupérer toute l'information que Paul a mais ne souhaitez pas l'avoir encore dans votre branche, vous pouvez lancer git fetch pb :

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

La branche master de Paul est accessible localement en tant que `pb/master` — vous pouvez la fusionner dans une de vos propres branches, ou vous pouvez extraire une branche localement si vous souhaitez l'inspecter.

2.5.3 Récupérer et tirer depuis des dépôts distants

Comme vous venez tout juste de le voir, pour obtenir les données des dépôts distants, vous pouvez lancer :

```
$ git fetch [nom-distant]
```

Cette commande s'adresse au dépôt distant et récupère toutes les données de ce projet que vous ne possédez pas déjà. Après cette action, vous possédez toutes les références à toutes les branches contenues dans ce dépôt, que vous pouvez fusionner ou inspecter à tout moment (nous reviendrons plus précisément sur les branches et leur utilisation au chapitre 3).

Si vous clonez un dépôt, le dépôt distant est automatiquement ajouté sous le nom origin. Donc, `git fetch origin` récupère tout ajout qui a été poussé vers ce dépôt depuis que vous l'avez cloné ou la dernière fois que vous avez récupéré les ajouts. Il faut noter que la commande fetch tire les données dans votre dépôt local mais sous sa propre branche — elle ne les fusionne pas automatiquement avec aucun de vos travaux ni ne modifie votre copie de travail. Vous devez volontairement fusionner ses modifications distantes dans votre travail lorsque vous le souhaitez.

Si vous avez créé une branche pour suivre l'évolution d'une branche distante (Cf. la section suivante et le chapitre 3 pour plus d'information), vous pouvez utiliser la commande `git pull` qui récupère et fusionne automatiquement une branche distante dans votre branche locale. Ce comportement peut correspondre à une méthode de travail plus confortable, sachant que par défaut la commande `git clone` paramètre votre branche locale pour qu'elle suive la branche master du dépôt que vous avez cloné (en supposant que le dépôt distant ait une branche master). Lancer `git pull` récupère généralement les données depuis le serveur qui a été initialement cloné et essaie de la fusionner dans votre branche de travail actuel.

2.5.4 Pousser son travail sur un dépôt distant

Lorsque votre dépôt vous semble prêt à être partagé, il faut le pousser en amont. La commande pour le faire est simple : `git push [nom-distant] [nom-de-branche]`. Si vous souhaitez pousser votre branche master vers le serveur `origin` (pour rappel, cloner un dépôt définit automatiquement ces noms pour vous), alors vous pouvez lancez ceci pour pousser votre travail vers le serveur amont :

```
$ git push origin master
```

Cette commande ne fonctionne que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Si vous et quelqu'un d'autre clonez un dépôt au même moment et que cette autre personne pousse ses modifications et qu'après vous tentez de pousser les vôtres, votre poussée sera rejetée à juste titre. Vous devrez tout d'abord tirer les modifications de l'autre personne et les fusionner avec les vôtres avant de pouvoir pousser. Référez-vous au chapitre 3 pour de plus amples informations sur les techniques pour pousser vers un serveur distant.

2.5.5 Inspecter un dépôt distant

Si vous souhaitez visualiser plus d'informations à propos d'un dépôt distant particulier, vous pouvez utiliser la commande `git remote show [nom-distant]`. Si vous lancez cette commande avec un nom court particulier, tel que `origin`, vous obtenez quelque chose comme :

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
```

```
    ticgit
```

Cela donne le liste des URL pour le dépôt distant ainsi que la liste des branches distantes suivies. Cette commande vous informe que si vous êtes sur la branche master et si vous lancez `git pull`, il va automatiquement fusionner la branche master du dépôt distant après avoir récupéré toutes les références sur le serveur distant. Cela donne aussi la liste des autres références qu'il aura tirées.

Le résultat ci-dessus est un exemple simple mais réaliste de dépôt distant. Lors d'une utilisation plus intense de Git, la commande `git remote show` fournira beaucoup d'information :

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

Cette commande affiche les branches poussées automatiquement lorsqu'on lance `git push` dessus. Elle montre aussi les branches distantes qui n'ont pas encore été rapatriées, les branches distantes présentes localement mais effacées sur le serveur, et toutes les branches qui seront fusionnées quand on lancera `git pull`.

2.5.6 Retirer et déplacer des branches distantes

Si vous souhaitez renommer une référence, dans les versions récentes de Git, vous pouvez lancer `git remote rename` pour modifier le nom court d'un dépôt distant. Par exemple, si vous souhaitez renommer `pb` en `paul`, vous pouvez le faire avec `git remote rename` :

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Il faut mentionner que ceci modifie aussi les noms de branches distantes. Celle qui était référencée sous `pb/master` l'est maintenant sous `paul/master`.

Si vous souhaitez retirer une référence pour certaines raisons — vous avez changé de serveur ou vous n'utilisez plus ce serveur particulier, ou peut-être un contributeur a cessé de contribuer — vous pouvez utiliser `git remote rm` :

```
$ git remote rm paul
$ git remote
origin
```

2.6 Balisage

À l'instar de la plupart des VCS, Git donne la possibilité de baliser un certain état dans l'historique comme important. Généralement, les gens utilisent cette fonctionnalité pour marquer les états de publication (v1.0 et ainsi de suite). Dans cette section, nous apprendrons comment lister les différentes balises, comment créer de nouvelles balises et les différents types de balises.

2.6.1 Lister vos balises

Lister les balises existantes dans Git est très simple. Tapez juste `git tag` :

```
$ git tag
v0.1
v1.3
```

Cette commande liste les balises dans l'ordre alphabétique. L'ordre dans lequel elles apparaissent n'a aucun rapport avec l'historique.

Vous pouvez aussi rechercher les balises correspondant à un motif particulier. Par exemple, le dépôt des sources de Git contient plus de 240 balises. Si vous souhaitez ne visualiser que les série 1.4.2, vous pouvez lancer ceci :

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

2.6.2 Créer des balises

Git utilise deux types principaux de balises : légères et annotées. Une balise légère ressemble beaucoup à une branche qui ne change pas, c'est juste un pointeur sur un commit spécifique. Les balises annotées, par contre sont stockées en tant qu'objets à part entière dans la base de données de Git. Elles ont une somme de contrôle, contiennent le nom et l'adresse e-mail du créateur, la date, un message de balisage et peuvent être signées et vérifiées avec GNU Privacy Guard (GPG). Il est

généralement recommandé de créer des balises annotées pour générer toute cette information mais si la balise doit rester temporaire ou l'information supplémentaire n'est pas désirée, il reste toujours les balises légères.

2.6.3 Les balises annotées

Créer des balises annotées est simple avec Git. Le plus simple est de spécifier l'option `-a` à la commande `tag` :

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

L'option `-m` permet de spécifier le message de balisage qui sera stocké avec la balise. Si vous ne spécifiez pas de message en ligne pour une balise annotée, Git lance votre éditeur pour pouvoir le saisir.

Vous pouvez visualiser les données de la balise à côté du commit qui a été marqué en utilisant la commande `git show` :

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Cette commande affiche le nom du créateur, la date de création de la balise et le message d'annotation avant de montrer effectivement l'information de validation.

2.6.4 Les balises signées

Vous pouvez aussi signer vos balises avec GPG, à condition d'avoir une clé privée. Il suffit de spécifier l'option `-s` au lieu de `-a` :

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

En lançant `git show` sur cette balise, on peut visualiser la signature GPG attachée :

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Plus loin, nous verrons comment vérifier une balise signée.

2.6.5 Les balises légères

Une autre manière de baliser les commits est d'utiliser les balises légères. Celles-ci se réduisent à stocker la somme de contrôle d'un commit dans un fichier, aucune autre information n'est conservée. Pour créer une balise légère, il suffit de n'utiliser aucune des option `-a`, `-s` ou `-m` :

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Cette fois-ci, en lançant `git show` sur la balise, on ne voit plus aucune information complémentaire. La commande ne montre que l'information de commit :

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

2.6.6 Vérifier des balises

Pour vérifier une balise signée, il faut utiliser `git tag -v [nom-de-balise]`. Cette commande utilise GPG pour vérifier la signature. La clé publique du signataire doit être présente dans votre trousseau :

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Si la clé publique du signataire n'est pas présente dans le trousseau, la commande donne le résultat suivant :

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

2.6.7 Baliser après coup

Vous pouvez aussi baliser des commits plus anciens. Supposons que l'historique des commits ressemble à ceci :

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Fusion branche 'experimental'
a6b4c97498bd301d84096da251c98a07c7723e65 Début de l'écriture support
0d52aaab4479697da7686c15f77a3d64d9165190 Un truc de plus
6d52a271eda8725415634dd79daabbc4d9b6008e Fusion branche 'experimental'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc ajout d'une fonction de validatn
4682c3261057305bdd616e23b64b0857d832627b ajout fichier afaire
166ae0c4d3f420721acbb115cc33848dfcc2121a début de l'ecriture support
9fceb02d0ae598e95dc970b74767f19372d61af8 mise à jour rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc validation afaire
8a5cbc430f1a9c3d00faaeffd07798508422908a mise à jour lisezmoi
```

Maintenant, supposons que vous avez oublié de baliser le projet à la version v1.2 qui correspondait au commit « mise à jour rakefile ». Vous pouvez toujours le faire après l'évènement. Pour baliser ce commit, vous spécifiez la somme de contrôle du commit (ou une partie) en fin de commande :

```
$ git tag -a v1.2 9fceb02
```

Le commit a été balisé :

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    mise à jour rakefile
...
```

2.6.8 Partager les balises

Par défaut, la commande `git push` ne transfère pas les balises vers les serveurs distants. Il faut explicitement pousser les balises après les avoir créées localement. Ce processus s'apparente à pousser des branches distantes – vous pouvez lancer `git push origin [nom-du-tag]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

Si vous avez de nombreuses balises que vous souhaitez pousser en une fois, vous pouvez aussi utiliser l'option `--tags` avec la commande `git push`. Ceci transférera toutes les nouvelles balises vers le serveur distant.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
 * [new tag]         v1.5 -> v1.5
```

A présent, lorsqu'une autre personne clone ou tire depuis votre dépôt, elle obtient aussi les balises.

2.7 Trucs et astuces

Avant de clore ce chapitre sur les bases de Git, voici quelques trucs et astuces qui peuvent rendre votre apprentissage de Git plus simple, facile ou familier. De nombreuses personnes utilisent parfaitement Git sans connaître aucun de ces trucs, et nous n'y ferons pas référence, ni ne considérerons leur connaissance comme des pré-requis pour la suite de ce livre, mais il est préférable de les connaître.

2.7.1 Auto-Complétion

Si vous utilisez le shell Bash, Git est livré avec un script d'auto-complétion utile. Téléchargez le code source de Git, et jetez un œil dans le répertoire `contrib/completion`. Il devrait y avoir un fichier nommé `git-completion.bash`. Copiez ce fichier dans votre répertoire personnel et ajoutez cette ligne à votre fichier `.bashrc` :

```
source ~/.git-completion.bash
```

Si vous souhaitez paramétrer Bash pour activer la complétion automatique de Git pour tous les utilisateur, copiez le script dans le répertoire `/opt/local/etc/bash_completion.d` sur les systèmes Mac ou dans le répertoire `/etc/bash_completion.d` sur les systèmes Linux. C'est le répertoire dans lequel Bash lit pour fournir automatiquement la complétion en ligne de commande.

Si vous utilisez Windows avec le Bash Git, qui est installé par défaut avec Git en msysGit, l'auto-complétion est pré-configurée.

Pressez la touche Tab lorsque vous écrivez une commande Git, et le shell devrait vous indiquer une liste de suggestions pour continuer la commande :

```
$ git co<tab><tab>
commit config
```

Dans ce cas, taper git co et appuyer sur la touche Tab deux fois suggère commit et config. Ajouter `m<tab>` complète `git commit` automatiquement.

Cela fonctionne aussi avec les options, ce qui est probablement plus utile. Par exemple, si vous tapez la commande `git log` et ne vous souvenez plus d'une des options, vous pouvez commencer à la taper, et appuyer sur la touche Tab pour voir ce qui peut correspondre :

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

C'est une astuce qui peut clairement vous éviter de perdre du temps ou de lire de la documentation.

2.7.2 Les alias Git

Git ne complète pas votre commande si vous ne la tapez que partiellement. Si vous ne voulez pas avoir à taper l'intégralité du texte de chaque commande, vous pouvez facilement définir un alias pour chaque commande en utilisant `git config`. Voici quelques exemples qui pourraient vous intéresser :

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Ceci signifie que, par exemple, au lieu de taper `git commit`, vous n'avez plus qu'à taper `git ci`. Au fur et à mesure de votre utilisation de git, vous utiliserez probablement d'autres commandes plus fréquemment. Dans ce cas, n'hésitez pas à créer de nouveaux alias.

Cette technique peut aussi être utile pour créer des commandes qui vous manquent. Par exemple, pour corriger le problème d'ergonomie que vous avez rencontré lors de la désindexation d'un fichier, vous pourriez créer un alias pour désindexer :

```
$ git config --global alias.unstage 'reset HEAD --'
```

Cela rend les deux commandes suivantes équivalentes :

```
$ git unstage fichierA
$ git reset HEAD fichierA
```

Cela rend les choses plus claires. Il est aussi commun d'ajouter un alias `last`, de la manière suivante :

```
$ git config --global alias.last 'log -1 HEAD'
```

Ainsi, vous pouvez visualiser plus facilement le dernier commit :

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

Pour explication, Git remplace simplement la nouvelle commande par tout ce que vous lui aurez demandé d'aliaser. Si par contre vous souhaitez lancer une commande externe plutôt qu'une sous-commande Git, vous pouvez commencer votre commande par un caractère `!`. C'est utile si vous écrivez vos propres outils pour travailler dans un dépôt Git. On peut par exemple aliaser `git visual` pour lancer `gitk` :

```
$ git config --global alias.visual "!gitk"
```

2.8 Résumé

A présent, vous pouvez réaliser toutes les opérations locales de base de Git — créer et cloner un dépôt, faire des modifications, les indexer et les valider, visualiser l'historique de ces modifications. Au prochain chapitre, nous traiterons de la fonctionnalité unique de Git : son modèle de branches.

Chapitre 3

Les branches avec Git

Quasiment tous les VCS ont une forme ou une autre de gestion de branche. Faire une branche signifie diverger de la ligne principale de développement et continuer à travailler sans se préoccuper de cette ligne principale. Dans de nombreux outils de gestion de version, cette fonctionnalité est souvent chère en ressources et nécessite souvent de créer une nouvelle copie du répertoire de travail, ce qui peut prendre longtemps dans le cas de grands projets.

De nombreuses personnes font référence au modèle de gestion de branche de Git comme LA fonctionnalité et c'est sûrement la spécificité de Git par rapport à la communauté des gestionnaires de version. Pourquoi est-elle si spéciale ? La méthode de Git pour gérer les branches est particulièrement légère, permettant de réaliser des embranchements quasi instantanément et de basculer de branche généralement aussi rapidement. À la différence de nombreux autres gestionnaires de version, Git encourage à travailler avec des méthodes qui privilégient la création et la fusion de branches, jusqu'à plusieurs fois par jour. Bien comprendre et maîtriser cette fonctionnalité est un atout pour faire de Git un outil unique qui peut littéralement changer la manière de développer.

3.1 Ce qu'est une branche

Pour réellement comprendre comment Git gère les branches, nous devons revenir en arrière et examiner de plus près comment Git stocke ses données. Comme vous pouvez vous en souvenir du chapitre 1, Git ne stocke pas ses données comme une série de changesets ou deltas, mais comme une série d'instantanés.

Lorsqu'on valide dans Git, Git stocke un objet commit qui contient un pointeur vers l'instantané du contenu qui a été indexé, les méta-données d'auteur et de message et zéro ou plusieurs pointeurs vers le ou les commits qui sont les parents directs de ce commit : zéro parent pour la première validation, un parent pour un commit normal et des parents multiples pour des commits qui sont le résultat de la fusion d'une ou plusieurs branches.

Pour visualiser ce concept, supposons un répertoire contenant trois fichiers, ces trois fichiers étant indexés puis validés. Indexer les fichiers signifie calculer la somme de contrôle pour chacun (la fonction de hachage SHA-1 mentionnée au chapitre 1), stocker cette version du fichier dans le dépôt Git (Git les nomme blobs) et ajouter la somme de contrôle à la zone d'index :

```
$ git add LISEZMOI test.rb LICENCE
$ git commit -m 'commit initial de mon projet'
```

Lorsque vous créez le commit en lançant la commande `git commit`, Git calcule la somme de contrôle de chaque répertoire (ici, seulement pour le répertoire racine) et stocke ces objets arbres dans le dépôt Git. Git crée alors un objet commit qui contient les méta-données et un pointeur vers l'arbre projet d'origine de manière à pouvoir recréer l'instantané si besoin.

Votre dépôt Git contient à présent cinq objets : un blob pour le contenu de chacun des trois fichiers, un arbre qui liste les contenus des répertoires et spécifie quels noms de fichier sont attachés à quels blobs et un objet commit avec le pointeur vers l'arbre d'origine et toutes les méta-données attachées au commit. Conceptuellement, les données contenues dans votre dépôt git ressemblent à la Figure 3.1.

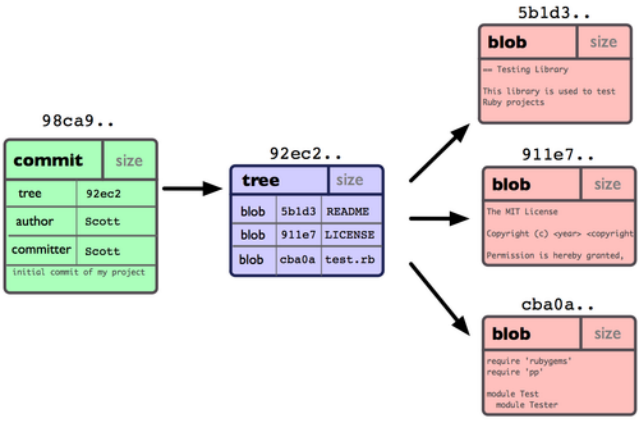

Figure 3.1: *Données d'un unique commit.*

Si vous réalisez des modifications et validez à nouveau, le prochain commit stocke un pointeur vers le commit immédiatement précédent. Après deux autres validations, l'historique pourrait ressembler à la figure 3-2.

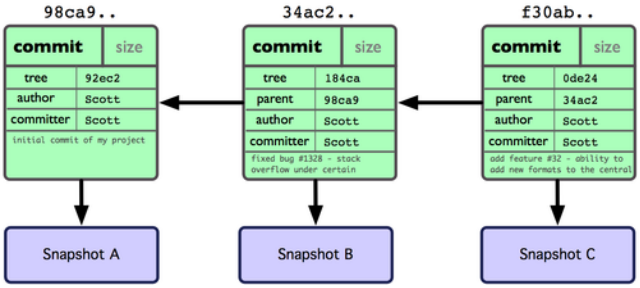

Figure 3.2: *Données et objets Git pour des validations multiples.*

Une branche dans Git est tout simplement un pointeur mobile léger vers un de ces objets commit. La branche par défaut dans Git s'appelle master. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. À chaque validation, le pointeur de la branche master avance automatiquement.

Que se passe-t-il si vous créez une nouvelle branche ? Et bien, cela crée un nouveau pointeur à déplacer. Supposons que vous créez une nouvelle branche nommée testing. Vous utilisez la commande `git branch` :

```
$ git branch testing
```

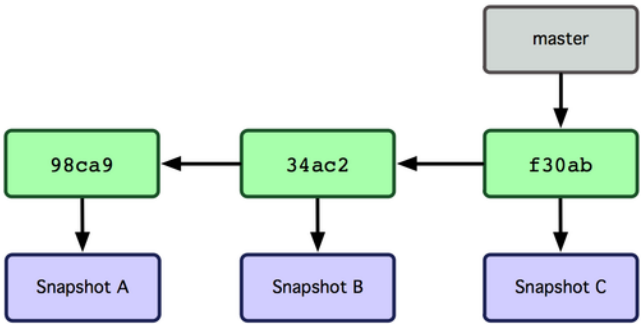

Figure 3.3: *Branche pointant dans l'historique des données de commit.*

Cela crée un nouveau pointeur vers le commit actuel (Cf. figure 3-4).

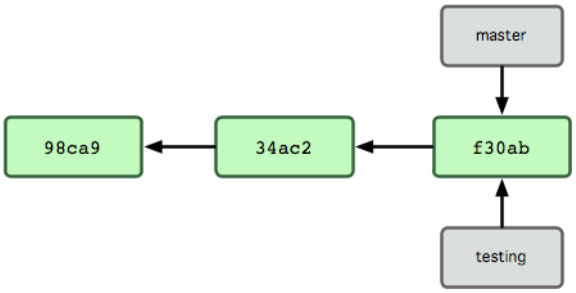

Figure 3.4: *Branches multiples pointant dans l'historique des données de commit.*

Comment Git connaît-il la branche sur laquelle vous vous trouvez ? Il conserve un pointeur spécial appelé HEAD. Remarquez que sous cette appellation se cache un concept très différent de celui utilisé dans les autres VCS tels que Subversion ou CVS. Dans Git, c'est un pointeur sur la branche locale où vous vous trouvez. Dans notre cas, vous vous trouvez toujours sur master. La commande git branch n'a fait que créer une nouvelle branche — elle n'a pas fait basculer la copie de travail vers cette branche (Cf. figure 3-5).

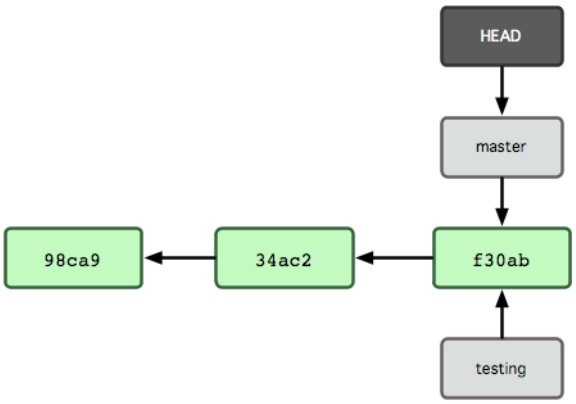

Figure 3.5: *fichier HEAD pointant sur la branche active*

Pour basculer vers une branche existante, il suffit de lancer la commande `git checkout`. Basculons vers la nouvelle branche testing :

```
$ git checkout testing
```

Cela déplace HEAD pour le faire pointer vers la branche testing (voir figure 3-6)

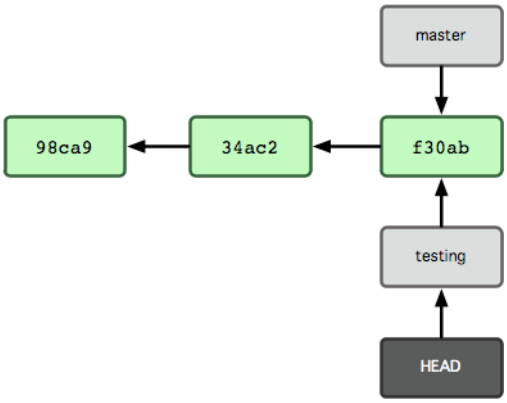

Figure 3.6: *HEAD pointe vers une autre branche quand on bascule de branche*

Qu'est-ce que cela signifie ? Et bien, faisons une autre validation :

```
$ vim test.rb
$ git commit -a -m 'petite modification'
```

La figure 3-7 illustre le résultat.

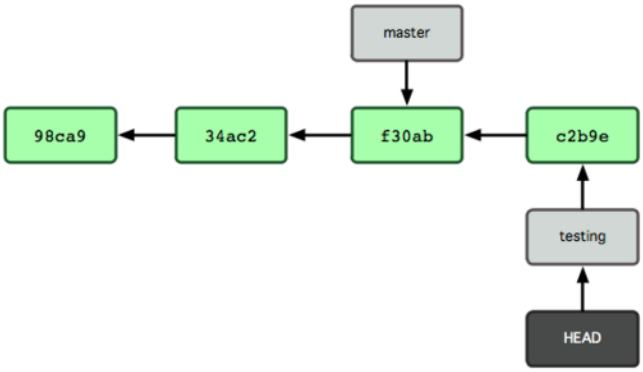

Figure 3.7: *La branche sur laquelle HEAD pointe avance avec chaque nouveau commit.*

C'est intéressant parce qu'à présent, votre branche testing a avancé, tandis que la branche master pointe toujours sur le commit sur lequel vous étiez lorsque vous avez lancé `git checkout` pour basculer de branche. Retournons sur la branche master :

```
$ git checkout master
```

La figure 3-8 montre le résultat.

Cette commande a réalisé deux actions. Elle a remis le pointeur HEAD sur la branche master et elle a replacé les fichiers de la copie de travail dans l'état pointé par master. Cela signifie aussi que les modifications que vous réalisez à partir de maintenant divergeront de l'ancienne version du projet. Cette commande retire les modifications réalisées dans la branche testing pour vous permettre de repartir dans une autre direction de développement.

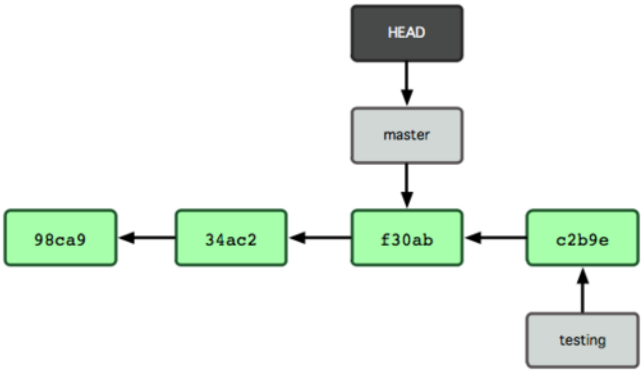

Figure 3.8: *HEAD se déplace sur une autre branche lors d'un checkout.*

Réalisons quelques autres modifications et validons à nouveau :

```
$ vim test.rb
$ git commit -a -m 'autres modifications'
```

Maintenant, l'historique du projet a divergé (voir figure 3-9). Vous avez créé une branche et basculé dessus, avez réalisé des modifications, puis avez rebasculé sur la branche principale et réalisé d'autres modifications. Ces deux modifications sont isolées dans des branches séparées. Vous pouvez basculer d'une branche à l'autre et les fusionner quand vous êtes prêt. Vous avez fait tout ceci avec de simples commandes `branch` et `checkout`.

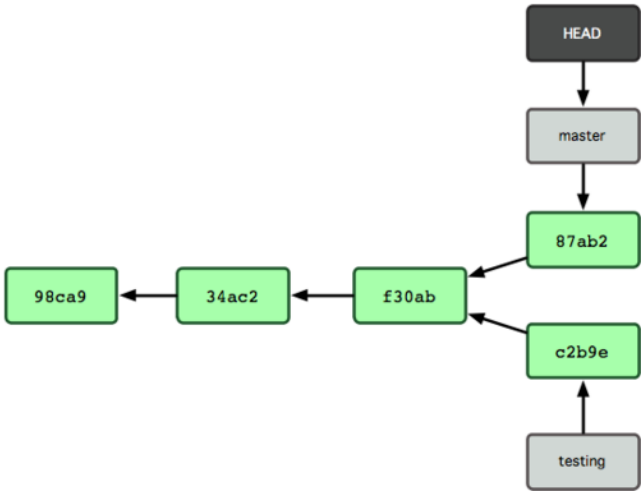

Figure 3.9: *Les historiques de branche ont divergé.*

Parce que dans Git, une branche n'est en fait qu'un simple fichier contenant les 40 caractères de la somme de contrôle SHA-1 du commit sur lequel elle pointe, les branches ne coûtent rien à créer et détruire. Créer une branche est aussi rapide qu'écrire un fichier de 41 caractères (40 caractères plus un retour chariot).

C'est une différence de taille avec la manière dont la plupart des VCS gèrent les branches, qui implique de copier tous les fichiers du projet dans un second répertoire. Cela peut durer plusieurs secondes ou même quelques minutes selon la taille du projet, alors que pour Git, le processus est toujours instantané. De plus, comme nous enregistrons les parents quand nous validons les modifications, la détermination de l'ancêtre commun pour la fusion est réalisée automatiquement et

de manière très facile. Ces fonctionnalités encouragent naturellement les développeurs à créer et utiliser souvent des branches.

Voyons pourquoi vous devriez en faire autant.

3.2 Brancher et fusionner : les bases

Suivons un exemple simple de branche et fusion dans une utilisation que vous feriez dans le monde réel. Vous feriez les étapes suivantes :

1. Travailler sur un site web
2. Créer une branche pour une nouvelle Story sur laquelle vous souhaiteriez travailler
3. Réaliser quelques tâches sur cette branche

À cette étape, vous recevez un appel pour vous dire qu'un problème critique a été découvert et qu'il faut le régler au plus tôt. Vous feriez ce qui suit :

1. Revenir à la branche de production
2. Créer une branche et y développer le correctif
3. Après un test, fusionner la branche de correctif et pousser le résultat à la production
4. Rebasculer à la branche initiale et continuer le travail

3.2.1 Le branchement de base

Premièrement, supposons que vous êtes à travailler sur votre projet et avez déjà quelques commits (voir figure 3-10).

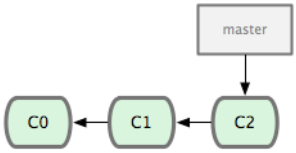

Figure 3.10: *Un historique simple et court.*

Vous avez décidé de travailler sur le problème numéroté #53 dans le suivi de faits techniques que votre entreprise utilise. Pour clarifier, Git n'est pas lié à un gestionnaire particulier de faits techniques. Mais comme le problème #53 est un problème ciblé sur lequel vous voulez travailler, vous allez créer une nouvelle branche dédiée à sa résolution. Pour créer une branche et y basculer tout de suite, vous pouvez lancer la commande `git checkout` avec l'option `-b` :

```
$ git checkout -b prob53
Switched to a new branch "prob53"
```

C'est un raccourci pour :

```
$ git branch prob53
$ git checkout prob53
```

La figure 3-11 illustre le résultat.

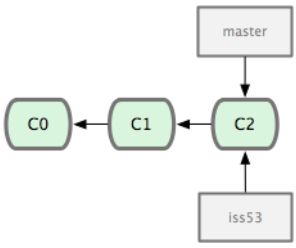

Figure 3.11: *Création d'un nouveau pointeur de branche.*

Vous travaillez sur votre site web et validez des modifications. Ce faisant, la branche `prob53` avance, parce que vous l'avez extraite (c'est-à-dire que votre pointeur HEAD pointe dessus, voir figure 3-12) :

```
$ vim index.html
$ git commit -a -m 'ajout d'un pied de page [problème 53]'
```

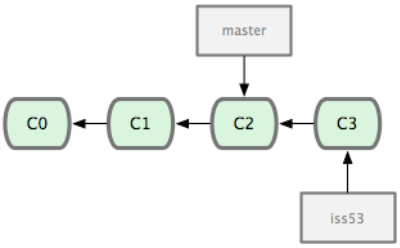

Figure 3.12: *La branche prob53 a avancé avec votre travail.*

Maintenant vous recevez un appel qui vous apprend qu'il y a un problème sur le site web, un problème qu'il faut résoudre immédiatement. Avec Git, vous n'avez pas besoin de déployer les modifications déjà validée pour `prob53` avec les correctifs du problème et vous n'avez pas non plus à suer pour éliminer ces modifications avant de pouvoir appliquer les correctifs du problème en production. Tout ce que vous avez à faire, c'est simplement rebasculer sur la branche `master`.

Cependant, avant de le faire, notez que si votre copie de travail ou votre zone de préparation contient des modifications non validées qui sont en conflit avec la branche que vous extrayez, Git ne vous laissera pas basculer de branche. Le mieux est d'avoir votre copie de travail dans un état propre au moment de basculer de branche. Il y a des moyens de contourner ceci (précisément par la planque et l'amendement de commit) dont nous parlerons plus loin. Pour l'instant, vous avez validé tous vos changements dans la branche `prob53` et vous pouvez donc rebasculer vers la branche `master` :

```
$ git checkout master
Switched to branch "master"
```

À présent, votre répertoire de copie de travail est exactement dans l'état précédent les modifications pour le problème #53 et vous pouvez vous consacrer à votre correctif. C'est un point important : Git réinitialise le répertoire de travail pour qu'il ressemble à l'instantané de la validation sur laquelle la branche que vous extrayez pointe. Il ajoute, retire et modifie les fichiers automatiquement pour assurer que la copie de travail soit identique à ce qu'elle était lors de votre dernière validation sur la branche.

Ensuite, vous avez un correctif à faire. Créons une branche de correctif sur laquelle travailler jusqu'à ce que ce soit terminé (voir figure 3-13) :

```
$ git checkout -b 'correctif'
Switched to a new branch "correctif"
$ vim index.html
$ git commit -a -m "correction d'une adresse mail incorrecte"
[correctif]: created 3a0874c: "correction d'une adresse mail incorrecte"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

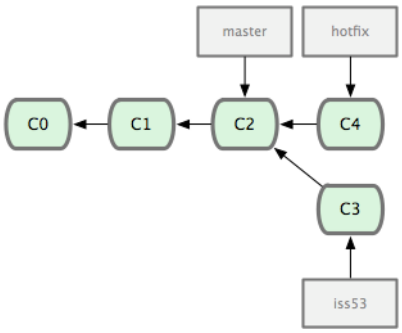

Figure 3.13: *Branche de correctif basée à partir de la branche master.*

Vous pouvez lancer vos tests, vous assurer que la correction est efficace et la fusionner dans la branche master pour la déployer en production. Vous réalisez ceci au moyen de la commande `git merge` :

```
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast forward
 LISEZMOI |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Vous noterez la mention « Fast forward » qui signifie avance rapide dans cette fusion. Comme le commit pointé par la branche que vous avez fusionné était directement descendant du commit sur lequel vous vous trouvez, Git a avancé le pointeur en avant. Autrement dit, lorsque l'on cherche

à fusionner un commit qui peut être joint en suivant l'historique depuis le commit d'origine, Git avance simplement le pointeur car il n'y a pas de travaux divergeant à réellement fusionner — ceci s'appelle l'avance rapide.

Votre modification est maintenant dans l'instantané du commit pointé par la branche `master` et vous pouvez déployer votre modification (voir figure 3-14)

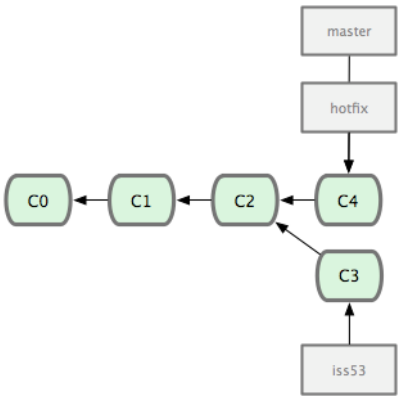

Figure 3.14: *Après la fusion, votre branche master pointe au même endroit que la correction.*

Après le déploiement de votre correction super-importante, vous voilà de nouveau prêt à travailler sur votre sujet précédent l'interruption. Cependant, vous allez avant tout effacer la branche `correctif` parce que vous n'en avez plus besoin et la branche `master` pointe au même endroit. Vous pouvez l'effacer avec l'option `-d` de la commande `git branch` :

```
$ git branch -d correctif
Deleted branch correctif (3a0874c).
```

Maintenant, il est temps de basculer sur la branche « travaux en cours » sur le problème #53 et de continuer à travailler dessus (voir figure 3-15) :

```
$ git checkout prob53
Switched to branch "prob53"
$ vim index.html
$ git commit -a -m 'Nouveau pied de page terminé [problème 53]'
[prob53]: created ad82d7a: "Nouveau pied de page terminé [problème 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Il est utile de noter que le travail réalisé dans `correctif` n'est pas contenu dans les fichiers de la branche `prob53`. Si vous avez besoin de les y rapatrier, vous pouvez fusionner la branche `master` dans la branche `prob53` en lançant la commande `git merge master`, ou vous pouvez retarder l'intégration de ces modifications jusqu'à ce que vous décidiez plus tard de rapatrier la branche `prob53` dans `master`.

3.2.2 Les bases de la fusion

Supposons que vous ayez décidé que le travail sur le problème #53 est terminé et se trouve donc prêt à être fusionné dans la branche `master`. Pour ce faire, vous allez rapatrier votre branche

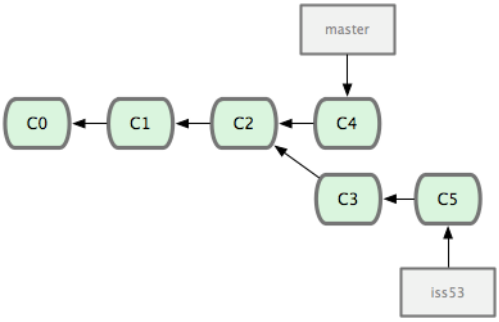

Figure 3.15: *Votre branche prob53 peut avancer indépendamment de master.*

`prob53` de la même manière que vous l'avez fait plus tôt pour la branche `correctif`. Tout ce que vous avez à faire est d'extraire la branche dans laquelle vous souhaitez fusionner et lancer la commande `git merge` :

```
$ git checkout master
$ git merge prob53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Le comportement semble légèrement différent de celui observé pour la fusion précédente de `correctif`. Dans ce cas, l'historique de développement a divergé à un certain point. Comme le commit sur la branche sur laquelle vous vous trouvez n'est plus un ancêtre direct de la branche que vous cherchez à fusionner, Git doit travailler. Dans ce cas, Git réalise une simple fusion à trois sources, en utilisant les deux instantanés pointés par les sommets des branches et l'ancêtre commun des deux. La figure 3-16 illustre les trois instantanés que Git utilise pour réaliser la fusion dans ce cas.

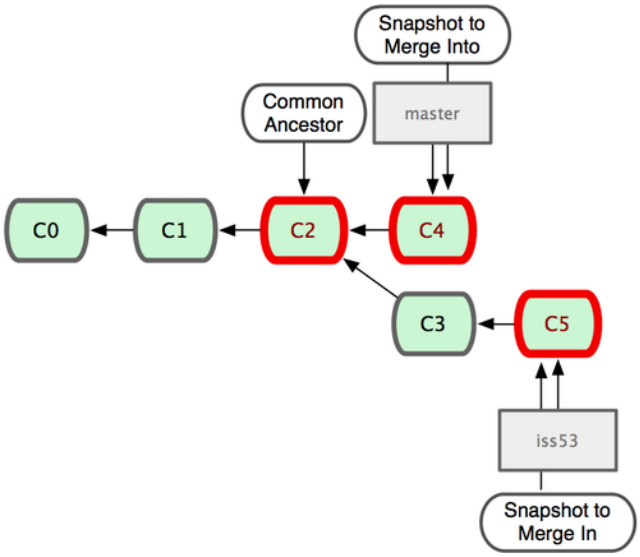

Figure 3.16: *Git identifie automatiquement la meilleure base d'ancêtre commun pour réaliser la fusion.*

Au lieu de simplement d'avancer le pointeur de branche, Git crée un nouvel instantané qui résulte de la fusion à trois branches et crée automatiquement un nouveau commit qui pointe dessus

(voir figure 3-17). On appelle ceci un commit de fusion, qui est spécial en ce qu'il comporte plus d'un parent.

Il est à noter que Git détermine par lui-même le meilleur ancêtre commun à utiliser comme base de fusion ; ce comportement est très différent de celui de CVS ou Subversion (antérieur à la version 1.5), où le développeur en charge de la fusion doit trouver par lui-même la meilleure base de fusion. Cela rend la fusion tellement plus facile dans Git que dans les autres systèmes.

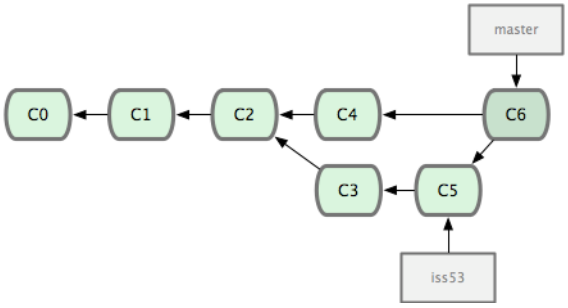

Figure 3.17: *Git crée automatiquement un nouvel objet commit qui contient le travail fusionné.*

A présent que votre travail a été fusionné, vous n'avez plus besoin de la branche `prob53`. Vous pouvez l'effacer et fermer manuellement le ticket dans votre outil de suivi de faits techniques :

```
$ git branch -d prob53
```

3.2.3 Conflits de fusion

Quelques fois, le processus ci-dessus ne se passe pas sans accroc. Si vous avez modifié différemment la même partie du même fichier dans les deux branches que vous souhaitez fusionner, Git ne sera pas capable de réaliser proprement la fusion. Si votre résolution du problème #53 a modifié la même section de fichier que le `correctif`, vous obtiendrez une conflit de fusion qui ressemble à ceci :

```
$ git merge prob53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git n'a pas automatiquement créé le commit du fusion. Il a arrêté le processus le temps que vous résolviez le conflit. Lancez `git status` pour voir à tout moment après l'apparition du conflit de fusion quels fichiers n'ont pas été fusionnés :

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
```

```
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# unmerged:   index.html
#
```

Tout ce qui comporte des conflits de fusion et n'a pas été résolu est listé comme `unmerged`. Git ajoute des marques de conflit standard dans les fichiers qui comportent des conflits, pour que vous puissiez les ouvrir et résoudre les conflits manuellement. Votre fichier contient des sections qui ressemblent à ceci :

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> prob53:index.html
```

Cela signifie que la version dans HEAD (votre branche master, parce que c'est celle que vous aviez extraite quand vous avez lancé votre commande de fusion) est la partie supérieure de ce bloc (tout ce qui se trouve au dessus de la ligne =======), tandis que la version de la branche `prob53` se trouve en dessous. Pour résoudre le conflit, vous devez choisir une partie ou l'autre ou bien fusionner leurs contenus par vous-même. Par exemple, vous pourriez choisir de résoudre ce conflit en remplaçant tout le bloc par ceci :

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Cette résolution comporte des parties de chaque section et les lignes <<<<<<<, ======= et >>>>>>> ont été complètement effacées. Après avoir résolu chacune de ces sections dans chaque fichier comportant un conflit, lancez `git add` sur chaque fichier pour le marquer comme résolu. Préparer le fichier en zone de préparation suffit à le marquer résolu pour Git. Si vous souhaitez utiliser un outil graphique pour résoudre ces problèmes, vous pouvez lancer `git mergetool` qui démarre l'outil graphique de fusion approprié et vous permet de naviguer dans les conflits :

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

Si vous souhaitez utiliser un outil de fusion autre que celui par défaut (Git a choisi `opendiff` pour moi dans ce cas car j'utilise la commande sous Mac), vous pouvez voir tous les outils supportés après l'indication « merge tool candidates ». Tapez le nom de l'outil que vous préféreriez utiliser. Au chapitre 7, nous expliquerons comment changer cette valeur par défaut dans votre environnement.

Après avoir quitté l'outil de fusion, Git vous demande si la fusion a été réussie. Si vous répondez par la positive à l'outil, il indexe le fichier pour le marquer comme résolu.

Vous pouvez lancer à nouveau la commande `git status` pour vérifier que tous les conflits ont été résolus :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   index.html
#
```

Cela vous convient et vous avez vérifié que tout ce qui comportait un conflit a été indexé, vous pouvez taper la commande `git commit` pour finaliser le commit de fusion. Le message de validation ressemble d'habitude à ceci :

```
Merge branch 'prob53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Vous pouvez modifier ce message pour inclure les détails sur la résolution du conflit si vous pensez que cela peut être utile lors d'une revue ultérieure — pourquoi vous avez fait ceci si ce n'est pas clair.

3.3 Gestion de branches

Après avoir créé, fusionné et effacé des branches, regardons de plus près les outils de gestion de branche qui s'avèreront utiles lors d'une utilisation intensive des branches.

La commande `git branch` fait plus que créer et effacer des branches. Si vous la lancez sans argument, vous obtenez la liste des branches courantes :

```
$ git branch
  prob53
* master
  testing
```

Notez le caractère `*` qui préfixe la branche `master`. Ce caractère indique la branche qui a été extraite. Ceci signifie que si vous validez des modifications, la branche `master` avancera avec votre travail. Pour visualiser les dernières validations sur chaque branche, vous pouvez lancer le commande `git branch -v` :

```
$ git branch -v
  prob53   93b412c fix javascript issue
* master  7a98805 Merge branch 'prob53'
  testing 782fd34 add scott to the author list in the readmes
```

Une autre option permettant de voir l'état des branches permet de filtrer cette liste par les branches qui ont ou n'ont pas encore été fusionnées dans la branche courante. Les options `--merged` et `--no-merge` sont disponibles depuis la version 1.5.6 de Git. Pour voir quelles branches ont déjà été fusionnées dans votre branche actuelle, lancez `git branch --merged` :

```
$ git branch --merged
  prob53
* master
```

Comme vous avez déjà fusionné `prob53` auparavant, vous la voyez dans votre liste. Les branches de cette liste qui ne comportent pas l'étoile en préfixe peuvent généralement être effacées sans risque au moyen de `git branch -d` ; vous avez déjà incorporé leurs modifications dans une autre branche et n'allez donc rien perdre.

Lancez `git branch --no-merged` pour visualiser les branches qui contiennent des travaux qui n'ont pas encore été fusionnés :

```
$ git branch --no-merged
  testing
```

Ceci montre votre autre branche. Comme elle contient des modifications qui n'ont pas encore été fusionnées, un essai d'effacement par `git branch -d` se solde par un échec :

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Si vous souhaitez réellement effacer cette branche et perdre ainsi le travail réalisé, vous pouvez forcer l'effacement avec l'option `-D`, comme l'indique justement le message.

3.4 Travailler avec les branches

Après avoir acquis les bases pour brancher et fusionner, que pouvons-nous ou devons-nous en faire ? Ce chapitre traite des différents styles de développement que cette gestion de branche légère permet de mettre en place, pour vous aider à décider d'en incorporer une dans votre cycle de développement.

3.4.1 Branches au long cours

Comme Git utilise une fusion à 3 branches, fusionner une branche dans une autre plusieurs fois sur une longue période est généralement facile. Cela signifie que vous pouvez travailler sur plusieurs branches ouvertes en permanence pendant plusieurs étapes de votre cycle de développement ; vous pouvez fusionner régulièrement certaines dans d'autres.

De nombreux développeurs utilisent Git avec une méthode qui utilise cette approche, telle que n'avoir que du code entièrement stable et testé dans la branche `master`, voire du code qui a été ou sera publié. Ils ont une autre branche en parallèle appelée develop qui, lorsqu'elle devient stable, peut être fusionnée dans `master`. Cette branche est utilisée pour tirer des branches spécifiques (branches avec une faible durée de vie, telles que notre branche `prob53`) quand elles sont prêtes, s'assurer qu'elles passent tous les tests et n'introduisent pas de bugs.

En réalité, nous parlons de pointeurs qui se déplacent le long des lignes des commits réalisés. Les branches stables sont plus en profondeur dans la ligne de l'historique des commits tandis que les branches des derniers développements sont plus en hauteur dans l'historique (voir figure 3-18).

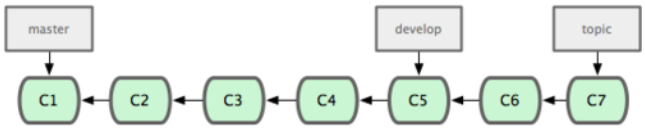

Figure 3.18: *Les branches les plus stables sont généralement plus bas dans l'historique des commits.*

C'est généralement plus simple d'y penser en terme de silos de tâches, où un ensemble de commits évolue vers un silo plus stable quand il a été complètement testé (voir figure 3-19).

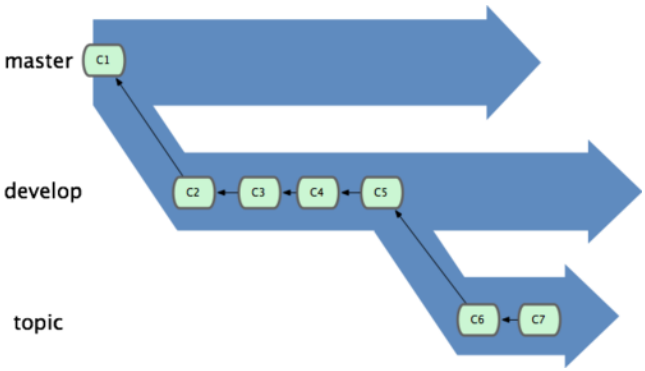

Figure 3.19: *Représentation des branches comme des silos.*

Vous pouvez reproduire ce schéma sur plusieurs niveaux de stabilité. Des projets plus gros ont aussi une branche `proposed` ou `pu` (proposed updates) qui permet d'intégrer des branches qui ne sont pas encore prêtes pour la prochaine version ou pour `master`. L'idée reste que les branches évoluent à différents niveaux de stabilité ; quand elles atteignent un niveau plus stable, elles peuvent être fusionnées dans la branche de stabilité supérieure. Une fois encore, les branches au long cours

ne sont pas nécessaires, mais s'avèrent souvent utiles, spécialement dans le cadre de projets gros ou complexes.

3.4.2 Les branches de sujet

Les branches de sujet sont tout de même utiles quelle que soit la taille du projet. Une branche de sujet est une branche de courte durée de vie créée et utilisée pour une fonctionnalité ou une tâche particulière. C'est une manière d'opérer que vous n'avez vraisemblablement jamais utilisée avec un autre VCS parce qu'il est généralement trop lourd de créer et fusionner des branches. Mais dans Git, créer, développer, fusionner et effacer des branches plusieurs fois par jour est monnaie courante.

Vous l'avez remarqué dans la section précédent avec les branches `prob53` et `correctif` que vous avez créées. Vous avez réalisé quelques validations sur elles et vous les avez effacées juste après les avoir fusionnées dans votre branche principale. Cette technique vous permet de basculer de contexte complètement et immédiatement. Il est beaucoup plus simple de réaliser des revues de code parce que votre travail est isolé dans des silos où toutes les modifications sont liées au sujet . Vous pouvez entreposer vos modifications ici pendant des minutes, des jours ou des mois, puis les fusionner quand elles sont prêtes, indépendamment de l'ordre dans lequel elles ont été créées ou de développées.

Supposons un exemple où pendant un travail (sur `master`), vous branchiez pour un problème (`prob91`), travaillez un peu dessus, vous branchiez une seconde branche pour essayer de trouver une autre manière de le résoudre (`prob91v2`), vous retourniez sur la branche `master` pour y travailler pendant un moment pour finalement brancher sur un dernière branche (`ideeidiote`) pour laquelle vous n'êtes pas sûr que ce soit une bonne idée. Votre historique de commit pourrait ressembler à la figure 3-20.

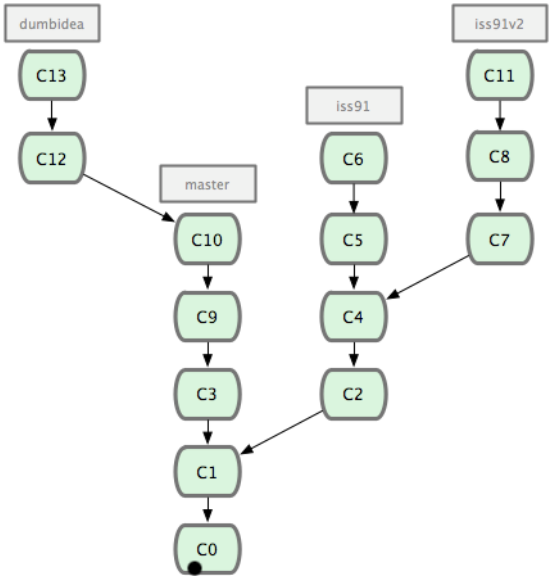

Figure 3.20: *Votre historique de commit avec de multiples branches de sujet.*

Maintenant, supposons que vous décidiez que vous préférez la seconde solution pour le problème (`prob91v2`) et que vous ayez montré la branche `ideeidiote` à vos collègues qui vous ont dit qu'elle était géniale. Vous pouvez jeter la branche `prob91` originale (en effaçant les commits C5 et C6) et fusionner les deux autres. Votre historique ressemble à présent à la figure 3-21.

Souvenez-vous que lors de la réalisation de ces actions, toutes ces branches sont complètement

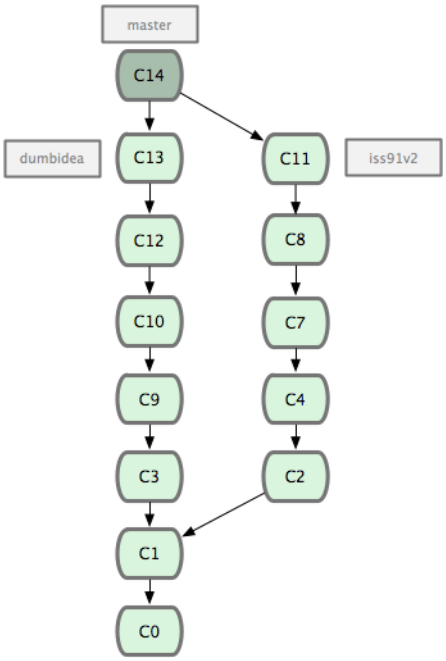

Figure 3.21: *Votre historique après la fusion de* `ideeidiote` *et* `prob91v2`.

locales. Lorsque vous branchez et fusionnez, tout est réalisé dans votre dépôt Git. Aucune communication avec un serveur n'a lieu.

3.5 Les branches distantes

Les branches distantes sont des références à l'état des branches sur votre dépôt distant. Ce sont des branches locales qu'on ne peut pas modifier ; elles sont modifiées automatiquement lors de communications réseau. Les branches distantes agissent comme des marques-pages pour vous aider à vous souvenir de l'état de votre dépôt distant lorsque vous vous y êtes connectés.

Elles prennent la forme de `(distant)/(branche)`. Par exemple, si vous souhaitiez visualiser l'état de votre branche `master` sur le dépôt distant `origin` lors de votre dernière communication, il vous suffit de vérifier la branche `origin/master`. Si vous étiez en train de travailler avec un collègue et qu'il a mis à jour la branche `prob53`, vous pourriez avoir votre propre branche `prob53` ; mais la branche sur le serveur pointerait sur le commit de `origin/prob53`.

Cela peut paraître déconcertant, alors éclaircissons les choses par un exemple. Supposons que vous avez un serveur Git sur le réseau à l'adresse `git.notresociete.com`. Si vous clonez à partir de ce serveur, Git le nomme automatiquement `origin` et en tire tout l'historique, crée un pointeur sur l'état actuel de la branche `master` et l'appelle localement `origin/master` ; vous ne pouvez pas la modifier. Git vous crée votre propre branche `master` qui démarre au même commit que la branche `master` d'origine, pour que vous puissiez commencer à travailler (voir figure 3-22).

Si vous travaillez sur votre branche locale `master` et que dans le même temps, quelqu'un pousse vers `git.notresociete.com` et met à jour cette branche, alors vos deux historiques divergent. Tant que vous restez sans contact avec votre serveur distant, votre pointeur `origin/master` n'avance pas (voir figure 3-23).

Lancez la commande `git fetch origin` pour synchroniser votre travail. Cette commande recherche le serveur hébergeant origin (dans notre cas, `git.notresociete.com`), en récupère toutes les nouvelles données et met à jour votre base de donnée locale en déplaçant votre pointeur

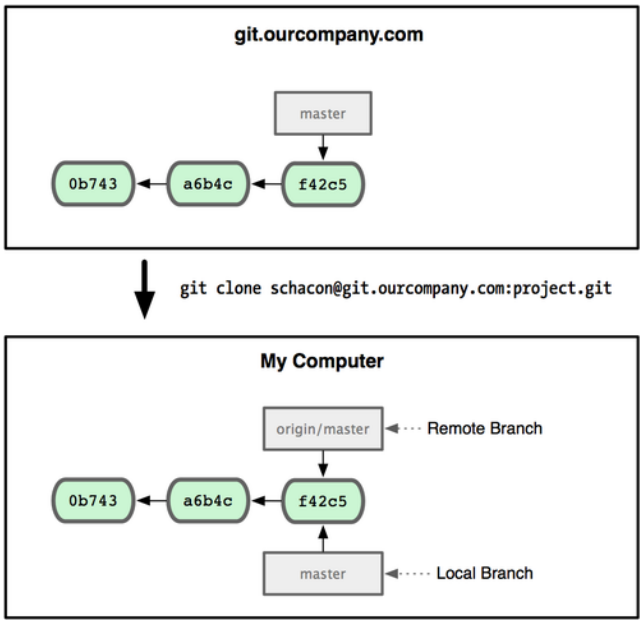

Figure 3.22: *Un clonage Git vous fournit une branche master et une branche origin/master pointant sur la branche master de l'origine.*

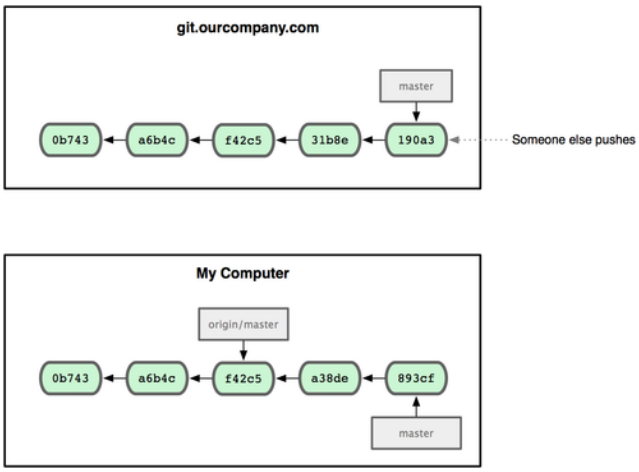

Figure 3.23: *Les travaux locaux et les modifications poussées sur le serveur distant font diverger les deux historiques.*

`origin/master` à sa valeur nouvelle à jour avec le serveur distant (voir figure 3-24).

Pour démontrer l'usage de multiples serveurs distants et le fonctionnement avec des branches multiples, supposons que vous avez un autre serveur Git interne qui n'est utilisé pour le développement que par une équipe. Ce serveur se trouve sur `git.equipe1.notresociete.com`. Vous pouvez l'ajouter aux références distantes de votre projet actuel en lançant la commande `git remote add` comme nous l'avons décrit au chapitre 2. Nommez ce serveur distant `equipeun` qui sera le raccourci pour l'URL complète (voir figure 3-25).

Maintenant, lancez `git fetch equipeun` pour récupérer l'ensemble des informations du serveur que vous ne possédez pas. Comme ce serveur contient déjà un sous-ensemble des données du serveur `origin`, Git ne récupère aucune donnée mais positionne une branche distante appelée `equipeun/master` qui pointe sur le commit que `equipeun` a comme branche `master` (voir figure 3-26).

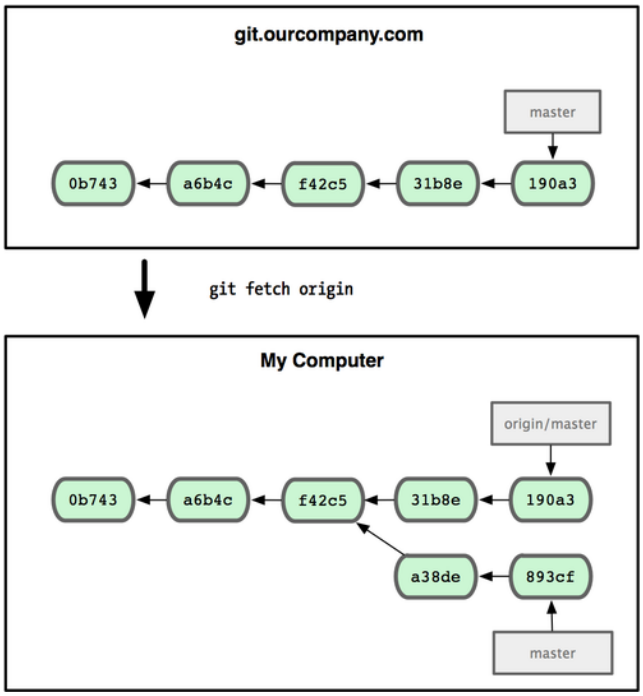

Figure 3.24: *La commande git fetch met à jour vos références distantes.*

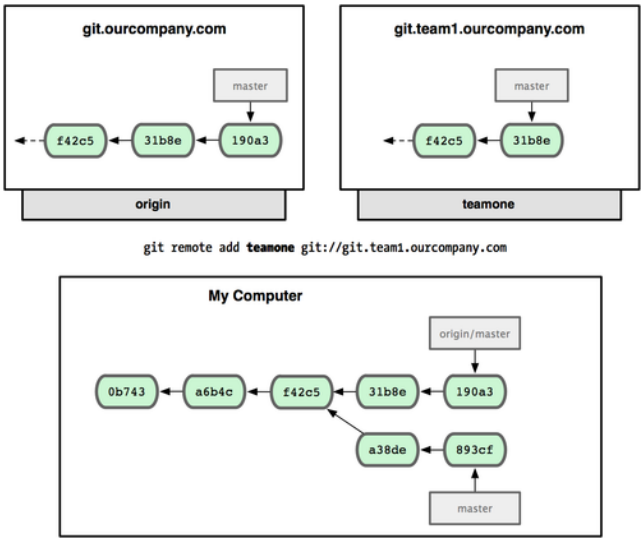

Figure 3.25: *Ajouter un autre serveur comme accès distant.*

3.5.1 Pousser vers un serveur

Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur le serveur distant sur lequel vous avez accès en écriture. Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants — vous devez pousser explicitement les branches que vous souhaitez partager. De cette manière, vous pouvez utiliser des branches privées pour le travail que vous ne souhaitez pas partager et ne pousser que les branches sur lesquelles vous souhaitez collaborer.

Si vous possédez une branche nommée `correctionserveur` sur laquelle vous souhaitez travailler avec des tiers, vous pouvez la pousser de la même manière que vous avez poussé votre première branche. Lancez `git push [serveur distant] [branche]` :

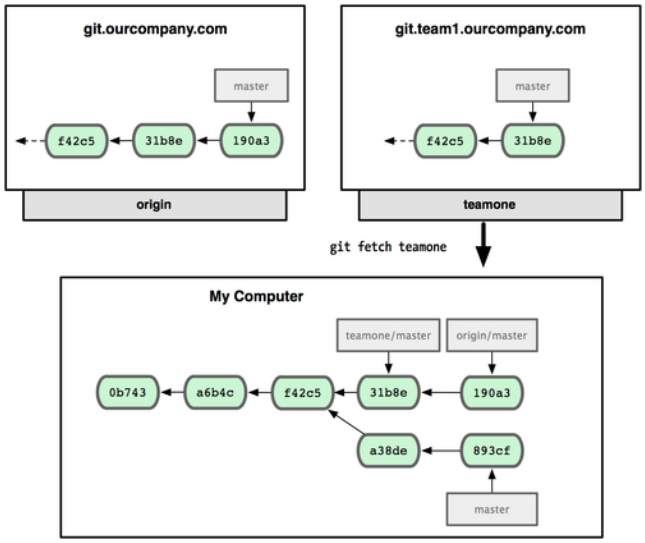

Figure 3.26: *Vous récupérez une référence locale à la branch master de equipeun.*

```
$ git push origin correctionserveur
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      correctionserveur -> correctionserveur
```

C'est un raccourci. En fait, Git étend le nom de branche `correctionserveur` en `refs/heads/correctionserveur:refs/heads/correctionserveur`, ce qui signifie « Prendre ma branche locale correctionserveur et la pousser pour mettre à jour la branche distante correctionserveur ». Nous traiterons plus en détail la partie `refs/heads/` au chapitre 9, mais vous pouvez généralement l'oublier. Vous pouvez aussi lancer `git push origin correctionserveur:correctionserveur`, qui réalise la même chose — ce qui signifie « Prendre ma branche correctionserveur et en faire la branche correctionserveur distante ». Vous pouvez utiliser ce format pour pousser une branche locale vers une branche distante nommée différemment. Si vous ne souhaitez pas l'appeler `correctionserveur` sur le serveur distant, vous pouvez lancer à la place `git push origin correctionserveur:branchegeniale` pour pousser votre branche locale `correctionserveur` sur la branche `branchegeniale` sur le projet distant.

La prochaine fois qu'un de vos collaborateurs récupère les données depuis le serveur, il récupérera une référence à l'état de la branche distante `origin/correctionserveur` :

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      correctionserveur    -> origin/correctionserveur
```

Important : lorsque l'on récupère une nouvelle branche depuis un serveur distant, il n'y a pas de création automatique d'une copie locale éditable. En d'autres termes, il n'y a pas de branche `correctionserveur`, seulement un pointeur sur la branche `origin/correctionserveur` qui n'est pas modifiable.

Pour fusionner ce travail dans votre branche actuelle de travail, vous pouvez lancer `git merge origin/correctionserveur`. Si vous souhaitez créer votre propre branche `correction-serveur` pour pouvoir y travailler, vous pouvez la baser sur le pointeur distant :

```
$ git checkout -b correctionserveur origin/correctionserveur
Branch correctionserveur set up to track remote branch refs/remotes/origin/
correctionserveur.
Switched to a new branch "correctionserveur"
```

Cette commande vous fournit une branche locale modifiable basée sur l'état actuel de `origin/correctionserveur`.

3.5.2 Suivre les branches

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement ce qu'on appelle une *branche de suivi*. Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante. Si vous vous trouvez sur une branche de suivi et que vous tapez `git push`, Git sélectionne automatiquement le serveur vers lequel pousser vos modifications. De même, `git pull` sur une de ces branches récupère toutes les références distantes et les fusionne automatiquement dans la branche distante correspondante.

Lorsque vous clonez un dépôt, il crée généralement automatiquement une branche `master` qui suit `origin/master`. C'est pourquoi les commandes `git push` et `git pull` fonctionnent directement sans plus de paramétrage. Vous pouvez néanmoins créer d'autres branches de suivi si vous le souhaitez, qui ne suivront pas `origin` ni la branche `master`. Un cas d'utilisation simple est l'exemple précédent, en lançant `git checkout -b [branche] [nomdistant]/[branche]`. Si vous avez Git version 1.6.2 ou plus, vous pouvez aussi utiliser l'option courte `--track` :

```
$ git checkout --track origin/correctionserveur
Branch correctionserveur set up to track remote branch refs/remotes/origin/
correctionserveur.
Switched to a new branch "serverfix"
```

Pour créer une branche locale avec un nom différent de celui de la branche distante, vous pouvez simplement utiliser la première version avec un nom de branch locale différent :

```
$ git checkout -b sf origin/correctionserveur
Branch sf set up to track remote branch refs/remotes/origin/correctionserveur.
Switched to a new branch "sf"
```

À présent, votre branche locale sf poussera vers et tirera automatiquement depuis origin/correctionserveur.

3.5.3 Effacer des branches distantes

Supposons que vous en avez terminé avec une branche distante. Vous et vos collaborateurs avez terminé une fonctionnalité et l'avez fusionnée dans la branche `master` du serveur distant (ou la branche correspondant à votre code stable). Vous pouvez effacer une branche distante en utilisant la syntaxe plutôt obtuse `git push [nomdistant] :[branch]`. Si vous souhaitez effacer votre branche `correctionserveur` du serveur, vous pouvez lancer ceci :

```
$ git push origin :correctionserveur
To git@github.com:schacon/simplegit.git
 - [deleted]         correctionserveur
```

Boum ! Plus de branche sur le serveur. Vous souhaiterez sûrement corner cette page parce que vous aurez besoin de cette commande et il y a de fortes chances que vous en oubliez la syntaxe. Un moyen mnémotechnique est de l'associer à la syntaxe de la commande `git push [nomdistant] [branchelocale]:[branchedistante]` que nous avons utilisé précédemment. Si vous éliminez la partie `[branchelocale]`, cela signifie « ne rien prendre de mon côté et en faire `[branchedistante]` ».

3.6 Rebaser

Dans Git, il y a deux façons d'intégrer les modifications d'une branche dans une autre : en fusionnant `merge` et en rebasant `rebase`. Dans ce chapitre, vous apprendrez la signification de rebaser, comment le faire, pourquoi c'est un outil plutôt ébouriffant et dans quels cas il est déconseillé de l'utiliser.

3.6.1 Les bases

Si vous revenez à un exemple précédent du chapitre sur la fusion (voir la figure 3-27), vous remarquerez que votre travail a divergé et que vous avez ajouté de commits sur deux branches différentes.

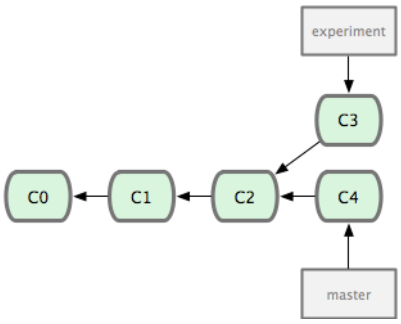

Figure 3.27: *Votre historique divergent initial.*

Comme nous l'avons déjà expliqué, le moyen le plus simple pour intégrer ensemble ces branches est la fusion via la commande `merge`. Cette commande réalise une fusion à trois branches entre les

deux derniers instantanés de chaque branche (C3 et C4) et l'ancêtre commun le plus récent (C2), créant un nouvel instantané (et un commit), comme montré par la figure 3-28.

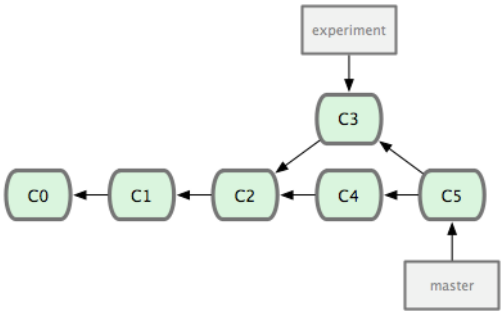

Figure 3.28: *Fusion d'une branche pour intégrer les historiques divergents.*

Cependant, il existe un autre moyen : vous pouvez prendre le patch de la modification introduite en C3 et le réappliquer sur C4. Dans Git, cette action est appelée *rebaser*. Avec la commande `rebase`, vous prenez toutes les modifications qui ont été validées sur une branche et vous les rejouez sur une autre.

Dans cet exemple, vous lanceriez les commandes suivantes :

```
$ git checkout experience
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Cela fonctionne en cherchant l'ancêtre commun le plus récent des deux branches (celle sur laquelle vous vous trouvez et celle sur laquelle vous rebasez), en récupérant toutes les différences introduites entre chaque validation de la branche sur laquelle vous êtes, en les sauvant dans des fichiers temporaires, en basculant sur la branche destination et en réappliquant chaque modification dans le même ordre. La figure 3-29 illustre ce processus.

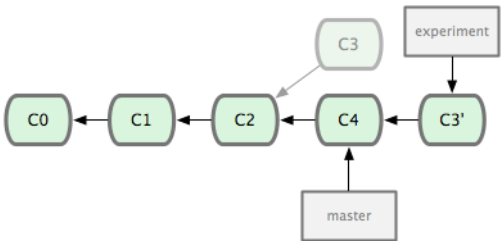

Figure 3.29: *Rebaser les modifications introduites par C3 sur C4.*

À ce moment, vous pouvez retourner sur la branche master et réaliser une fusion en avance rapide (voir figure 3-30).

À présent, l'instantané pointé par C3 est exactement le même que celui pointé par C5 dans l'exemple de fusion. Il n'y a pas de différence entre les résultats des deux types d'intégration, mais rebaser rend l'historique plus clair. Si vous examinez le journal de la branche rebasée, elle est devenue linéaire : toutes les modifications apparaissent en série même si elles ont eu lieu en parallèle.

Vous aurez souvent à rebaser pour vous assurer que les patchs que vous envoyez s'appliquent correctement sur une branche distante — par exemple, sur un projet où vous souhaitez contribuer

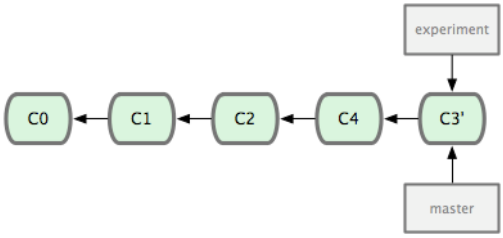

Figure 3.30: *Avance rapide sur la branche master.*

mais que vous ne maintenez pas. Dans ce cas, vous réaliseriez votre travail dans une branche puis vous rebaseriez votre travail sur `origin/master` quand vous êtes prêt à soumettre vos patches au projet principal. De cette manière, le mainteneur n'a pas à réaliser de travail d'intégration — juste une avance rapide ou simplement une application propre.

Il faut noter que l'instantané pointé par le commit final, qu'il soit le dernier des commits d'une opération de rebase ou le commit final issu d'une fusion, sont en fait le même instantané — c'est juste que l'historique est différent. Rebaser rejoue les modifications d'une ligne de commits sur une autre dans l'ordre d'apparition, alors que la fusion joint et fusionne les deux têtes.

3.6.2 Rebasages plus intéressants

Vous pouvez aussi faire rejouer votre rebasage sur autre chose qu'une branche. Prenez l'historique de la figure 3-31 par exemple. Vous avez créé une branche pour un sujet spécifique (`server`) pour ajouter des fonctionnalités côté serveur à votre projet et avez réalisé un commit. Ensuite, vous avez créé une branche pour ajouter des modifications côté client (`client`) et avez validé plusieurs fois. Finalement, vous avez rebasculé sur la branche `server` et avez réalisé quelques commits supplémentaires.

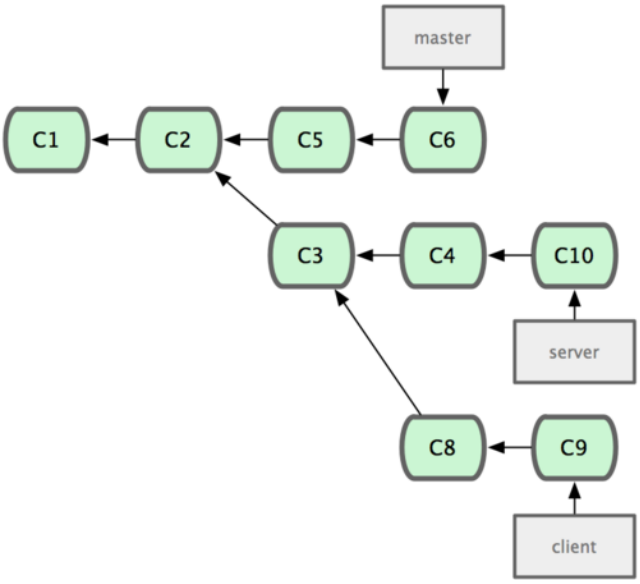

Figure 3.31: *Un historique avec une branche qui sort d'une autre branche de sujet.*

Supposons que vous décidez que vous souhaitez fusionner vos modifications pour le côté client dans votre ligne principale pour une publication mais vous souhaitez retenir les modifications pour la partie serveur jusqu'à ce qu'elles soient un peu plus testées. Vous pouvez récupérer les modifications pour le côté client qui ne sont pas sur le serveur (C8 et C9) et les rejouer sur la branche master

en utilisant l'option `--onto` de `git rebase` :

```
$ git rebase --onto master server client
```

Cela signifie en essence « Extrait la branche client, détermine les patchs depuis l'ancêtre commun des branches `client` et `server` puis rejoue les sur `master` ». C'est assez complexe, mais le résultat visible sur la figure 3-32 est assez impressionnant.

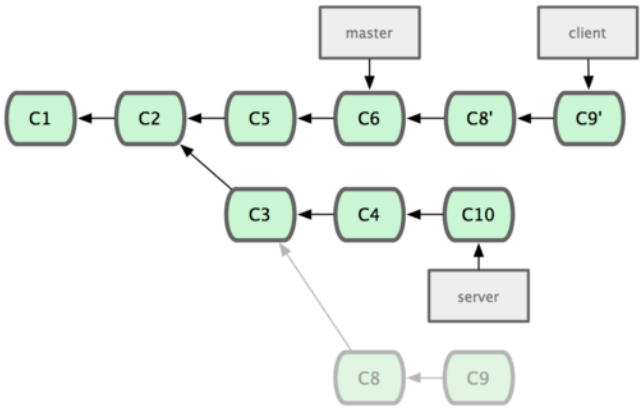

Figure 3.32: *Rebaser une branche de sujet sur une autre branche.*

Maintenant, vous pouvez faire une avance rapide sur votre branche master (voir figure 3-33) :

```
$ git checkout master
$ git merge client
```

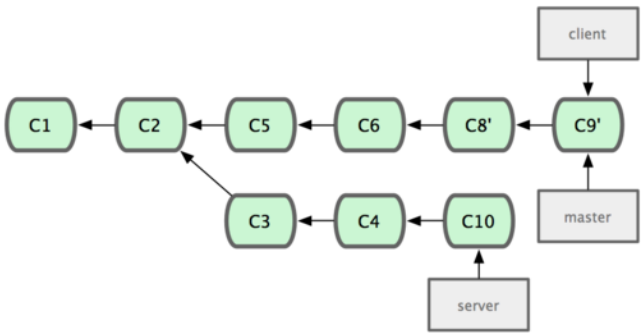

Figure 3.33: *Avance rapide sur votre branche master pour inclure les modifications de la branche client.*

Supposons que vous décidiez de tirer de votre branche server aussi. Vous pouvez rebaser la branche server sur la branche master sans avoir à l'extraire avant en utilisant `git rebase [branchedebase] [branchedesujet]` — qui extrait la branche de sujet (dans notre cas, `server`) pour vous et la rejoue sur la branche de base (`master`) :

```
$ git rebase master server
```

Cette commande rejoue les modifications de `server` sur le sommet de la branche `master`, comme indiqué dans la figure 3-34.

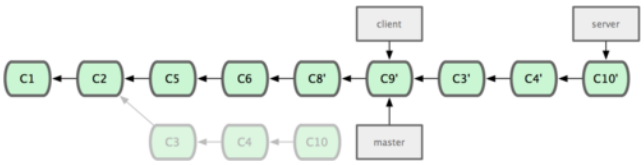

Figure 3.34: *Rebaser la branche server sur le sommet de la branche master.*

Ensuite, vous pouvez faire une avance rapide sur la branche de base (`master`) :

```
$ git checkout master
$ git merge server
```

Vous pouvez effacer les branches `client` et `server` une fois que tout le travail est intégré et que vous n'en avez plus besoin, éliminant tout l'historique de ce processus, comme visible sur la figure 3-35 :

```
$ git branch -d client
$ git branch -d server
```

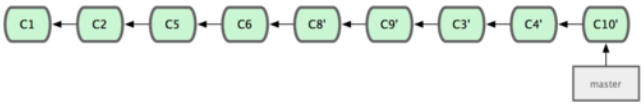

Figure 3.35: *L'historique final des commits.*

3.6.3 Les dangers de rebaser

Ah… mais les joies de rebaser ne viennent pas sans leurs contreparties, qui peuvent être résumées en une ligne :

Ne rebasez jamais des commits qui ont déjà été poussés sur un dépôt public.

Si vous suivez ce conseil, tout ira bien. Sinon, de nombreuses personnes vont vous haïr et vous serez méprisé par vos amis et votre famille.

Quand vous rebasez des données, vous abandonnez les commits existants et vous en créez de nouveaux qui sont similaires mais différents. Si vous poussez des commits quelque part et que d'autre les tirent et se basent dessus pour travailler et qu'après coup, vous réécrivez ces commits à l'aide de `git rebase` et les poussez à nouveau, vos collaborateurs devront re-fusionner leur travail et les choses peuvent rapidement devenir très désordonnées quand vous essaierez de tirer leur travail dans votre dépôt.

Examinons un exemple expliquant comment rebaser un travail déjà publié sur un dépôt public peut générer des gros problèmes. Supposons que vous clonez un dépôt depuis un serveur central et réalisez quelques travaux dessus. Votre historique de commits ressemble à la figure 3-36.

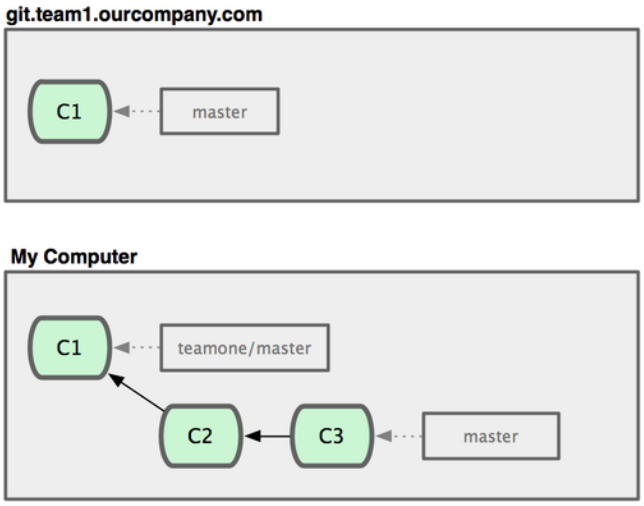

Figure 3.36: *Cloner un dépôt et baser du travail dessus.*

A présent, une autre personne travaille et inclut une fusion, puis elle pousse ce travail sur le serveur central. Vous le récupérez et vous fusionnez le nouvelle branche distante dans votre copie, ce qui donne l'historique de la figure 3-37.

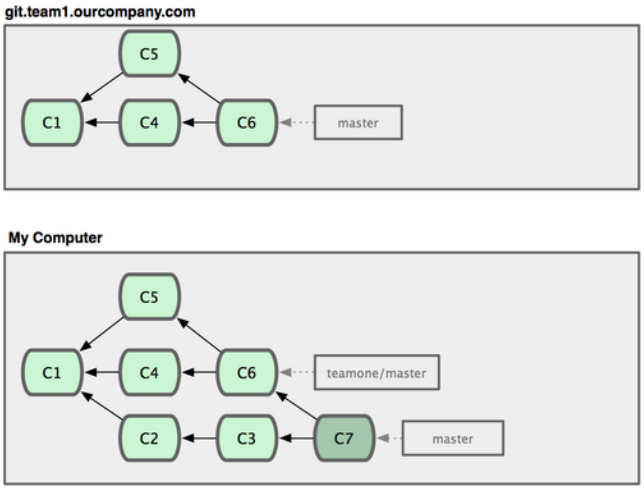

Figure 3.37: *Récupération de commits et fusion dans votre copie.*

Ensuite, la personne qui a poussé le travail que vous venez de fusionner décide de faire marche arrière et de rebaser son travail. Elle lance un `git push --force` pour forcer l'écrasement de l'historique sur le serveur. Vous récupérez alors les données du serveur, qui vous amènent les nouveaux commits.

A ce moment, vous devez fusionner son travail une nouvelle fois, même si vous l'avez déjà fait. Rebaser change les empreintes SHA-1 de ces commits, ce qui les rend nouveaux aux yeux de Git, alors qu'en fait, vous avez déjà le travail de C4 dans votre historique (voir figure 3-39).

Vous devez fusionner ce travail pour pouvoir continuer à suivre ce développeur dans le futur. Après fusion, votre historique contient à la fois les commits C4 et C4', qui ont des empreintes SHA-1 différentes mais introduisent les même modifications et ont les mêmes messages de validation. Si vous lancez `git log` lorsque votre historique ressemble à ceci, vous verrez deux commits qui ont la même date d'auteur et les mêmes messages, ce qui est déroutant. De plus, si vous poussez cet historique sur le serveur, vous réintroduirez tous ces commits rebasés sur le serveur central, ce qui

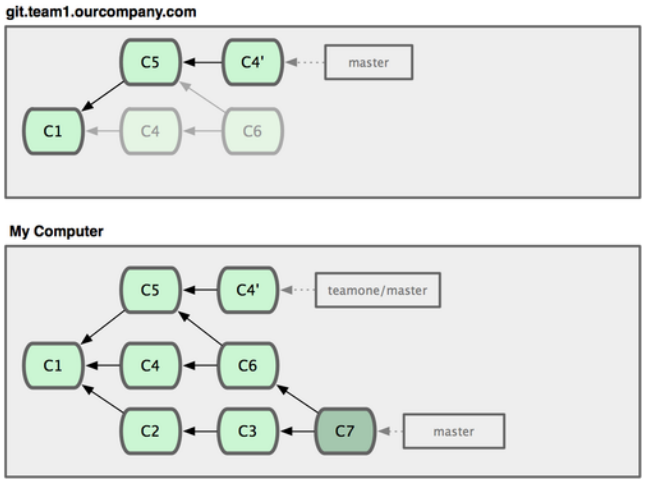

Figure 3.38: *Quelqu'un pousse des commits rebasés, en abandonnant les commits sur lesquels vous avez fondé votre travail.*

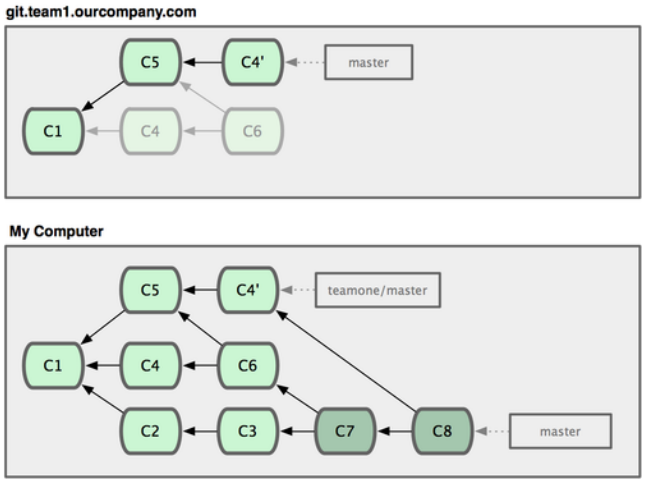

Figure 3.39: *Vous fusionnez le même travail une nouvelle fois dans un nouveau commit de fusion.*

va encore plus dérouter les autres développeurs.

Si vous considérez le fait de rebaser comme un moyen de nettoyer et réarranger des commits avant de les pousser et si vous vous en tenez à ne rebaser que des commits qui n'ont jamais été publiés, tout ira bien. Si vous tentez de rebaser des commits déjà publiés sur lesquels les gens ont déjà basé leur travail, vous allez au devant de gros problèmes énervants.

3.7 Résumé

Nous avons traité les bases des branches et de fusions dans Git. Vous devriez être à l'aise pour le création et le basculement sur de nouvelles branches, le basculement entre branches et la fusion de branches locales. Vous devriez aussi être capable de partager vos branches en les poussant sur une serveur partagé, travailler avec d'autres personnes sur des branches partagées et rebaser vos branches avant de les partager.

Chapitre 4

Git sur le serveur

À présent, vous devriez être capable de réaliser la plupart des tâches quotidiennes impliquant Git. Néanmoins, pour pouvoir collaborer avec d'autres personnes au moyen de Git, vous allez devoir disposer d'un dépôt distant Git. Bien que vous puissiez techniquement tirer des modifications et pousser des modification avec des dépôts individuels, cette pratique est découragée parce qu'elle introduit très facilement une confusion avec votre travail actuel. De plus, vous souhaitez que vos collaborateurs puissent accéder à votre dépôt de sources, y compris si vous n'êtes pas connecté — disposer d'un dépôt accessible en permanence peut s'avérer utile. De ce fait, la méthode canonique pour collaborer consiste à instancier un dépôt intermédiaire auquel tous ont accès, que ce soit pour pousser ou tirer. Nous nommerons ce dépôt le « serveur Git » mais vous vous apercevrez qu'héberger un serveur de dépôt Git ne consomme que peu de ressources et qu'en conséquence, on n'utilise que rarement une machine dédiée à cette tâche.

Un serveur Git est simple à lancer. Premièrement, vous devez choisir quels protocoles seront supportés. La première partie de ce chapitre traite des protocoles disponibles et de leurs avantages et inconvénients. La partie suivante explique certaines configurations typiques avec ces protocoles et comment les mettre en œuvre. Enfin, nous traiterons de quelques types d'hébergement, si vous souhaitez héberger votre code sur un serveur tiers, sans avoir à installer et maintenir un serveur par vous-même.

Si vous ne voyez pas d'intérêt à gérer votre propre serveur, vous pouvez sauter directement à la dernière partie de ce chapitre pour détailler les options pour mettre en place un compte hébergé, avant de continuer dans le chapitre suivant où les problématiques de développement distribué sont abordées.

Un dépôt distant est généralement un *dépôt nu* (*bare repository*), un dépôt Git qui n'a pas de copie de travail. Comme ce dépôt n'est utilisé que comme centralisateur de collaboration, il n'y a aucune raison d'extraire un instantané sur le disque ; seules les données Git sont nécessaires. Pour simplifier, un dépôt nu est le contenu du répertoire `.git` sans fioriture.

4.1 Protocoles

Git peut utiliser quatre protocoles réseau majeurs pour transporter des données : local, *Secure Shell* (SSH), Git et HTTP. Nous allons voir leur nature et dans quelles circonstances ils peuvent (ou ne peuvent pas) être utilisés.

Il est à noter que mis à part HTTP, tous le protocoles nécessitent l'installation de Git sur le serveur.

4.1.1 Protocole local

Le protocole de base est le protocole *local* pour lequel le dépôt distant est un autre répertoire dans le système de fichier. Il est souvent utilisé si tous les membres de l'équipe ont accès à un répertoire partagé via NFS par exemple ou dans le cas moins probable où tous les développeurs travaillent sur le même ordinateur. Ce dernier cas n'est pas optimum car tous les dépôts seraient hébergés de fait sur le même ordinateur, rendant ainsi toute défaillance catastrophique.

Si vous disposez d'un système de fichiers partagé, vous pouvez cloner, pousser et tirer avec un dépôt local. Pour cloner un dépôt ou pour l'utiliser comme dépôt distant d'un projet existant, utilisez le chemin vers le dépôt comme URL. Par exemple, pour cloner un dépôt local, vous pouvez lancer ceci :

```
$ git clone /opt/git/projet.git
```

Ou bien cela :

```
$ git clone file:///opt/git/projet.git
```

Git opère légèrement différemment si vous spécifiez explicitement le protocole `file://` au début de l'URL. Si vous spécifiez simplement le chemin, Git tente d'utiliser des liens durs ou une copie des fichiers nécessaires. Si vous spécifiez le protocole `file://`, Git lance un processus d'accès au travers du réseau, ce qui est généralement moins efficace. La raison d'utiliser spécifiquement le préfixe `file://` est la volonté d'obtenir une copie propre du dépôt, sans aucune référence ou aucun objet supplémentaire qui pourraient résulter d'un import depuis un autre système de gestion de version ou d'un action similaire (voir chapitre 9 pour les tâches de maintenance). Nous utiliserons les chemins normaux par la suite car c'est la méthode la plus efficace.

Pour ajouter un dépôt local à un projet Git existant, lancez ceci :

```
$ git remote add proj_local /opt/git/projet.git
```

Ensuite, vous pouvez pousser vers et tirer depuis ce dépôt distant de la même manière que vous le feriez pour un dépôt accessible sur le réseau.

Avantages

Les avantages des dépôts accessibles sur le système de fichier sont qu'ils sont simples et qu'ils utilisent les permissions du système de fichier. Si vous avez déjà un montage partagé auquel toute votre équipe a accès, déployer un dépôt est extrêmement facile. Vous placez la copie du dépôt nu à un endroit accessible de tous et positionnez correctement les droits de lecture/écriture de la même manière que pour tout autre partage. Nous aborderons la méthode pour exporter une copie de dépôt nu à cette fin dans la section suivante « Déployer Git sur un serveur ».

C'est un choix satisfaisant pour partager rapidement le travail. Si vous et votre coéquipier travaillez sur le même projet et qu'il souhaite partager son travail, lancer une commande telle que

`git pull /home/john/project` est certainement plus simple que de passer par un serveur intermédiaire.

Inconvénients

Les inconvénients de cette méthode sont qu'il est généralement plus difficile de rendre disponible un partage réseau depuis de nombreux endroits que de simplement gérer des accès réseau. Si vous souhaitez pousser depuis votre portable à la maison, vous devez monter le partage distant, ce qui peut s'avérer plus difficile et lent que d'accéder directement par un protocole réseau.

Il est aussi à mentionner que ce n'est pas nécessairement l'option la plus rapide à l'utilisation si un partage réseau est utilisé. Un dépôt local n'est rapide que si l'accès aux fichiers est rapide. Un dépôt accessible sur un montage NFS est souvent plus lent qu'un dépôt accessible via SSH sur le même serveur qui ferait tourner Git avec un accès au disques locaux.

4.1.2 Protocole SSH

Le protocole SSH est probablement le protocole de transport de Git le plus utilisé. Cela est dû au fait que l'accès SSH est déjà en place à de nombreux endroits et que si ce n'est pas le cas, cela reste très facile à faire. Cela est aussi dû au fait que SSH est le seul protocole permettant facilement de lire et d'écrire à distance. Les deux autres protocoles réseau (HTTP et Git) sont généralement en lecture seule et s'ils peuvent être utiles pour la publication, le protocole SSH est nécessaire pour les mises à jour de par ce qu'il permet l'écriture. SSH est un protocole authentifié suffisamment répandu et sa mise œuvre est simplifiée.

Pour cloner une dépôt Git à travers SSH, spécifiez le préfixe `ssh://` dans l'URL comme ceci :

```
$ git clone ssh://utilisateur@serveur:projet.git
```

ou ne spécifiez pas de protocole du tout — Git choisit SSH par défaut si vous n'êtes pas explicite :

```
$ git clone utilisateur@serveur:projet.git
```

Vous pouvez aussi ne pas spécifier de nom d'utilisateur et Git utilisera par défaut le nom de login.

Avantages

Les avantages liés à l'utilisation de SSH sont nombreux. Primo, vous ne pourrez pas faire autrement si vous souhaitez gérer un accès authentifié en écriture à votre dépôt au travers le réseau. Secundo, SSH est relativement simple à mettre en place, les daemons SSH sont facilement disponibles, les administrateurs réseaux sont habitués à les gérer et de nombreuses distributions de systèmes d'exploitation en disposent et proposent des outils de gestion. Ensuite, l'accès distant à travers SSH est sécurisé, toutes les données sont chiffrées et authentifiées. Enfin, comme les protocoles Git et local, SSH est efficace et permet de comprimer autant que possible les données avant de les transférer.

Inconvénients

Le point négatif avec SSH est qu'il est impossible de proposer un accès anonyme au dépôt. Les accès sont régis par les permission SSH, même pour un accès en lecture seule, ce qui s'oppose à une optique open-source. Si vous souhaitez utiliser Git dans un environnement d'entreprise, SSH peut bien être le seul protocole nécessaire. Si vous souhaitez proposer de l'accès anonyme en lecture seule à vos projets, vous aurez besoin de SSH pour vous permettre de pousser mais un autre protocole sera nécessaire pour permettre à d'autres de tirer.

4.1.3 Protocole Git

Vient ensuite le protocole Git. Celui-ci est géré par un *daemon* spécial livré avec Git. Ce démon écoute sur un port dédié (9418) et propose un service similaire au protocole SSH, mais sans aucune sécurisation. Pour qu'un dépôt soit publié via le protocole Git, le fichier `git-export-daemon-ok` doit exister mais mise à part cette condition sans laquelle le daemon refuse de publier un projet, il n'y a aucune sécurité. Soit le dépôt Git est disponible sans restriction en lecture, soit il n'est pas publié. Cela signifie qu'il ne permet de pousser des modifications. Vous pouvez activer la capacité à pousser mais étant donné l'absence d'authentification, n'importe qui sur internet peut pousser sur le dépôt. Autant dire que ce mode est rarement recherché.

Avantages

Le protocole Git est le protocole le plus rapide. Si vous devez servir un gros trafic pour un projet public ou un très gros projet qui ne nécessite pas d'authentification en lecture, il est très probable que vous devriez installer un *daemon* Git. Il utilise le même mécanisme de transfert de données que SSH, la surcharge du chiffrement et de l'authentification en moins.

Inconvénients

Le défaut du protocole Git est le manque d'authentification. N'utiliser que le protocole Git pour accéder à un projet n'est généralement pas suffisant. Il faut le coupler avec un accès SSH pour quelques développeurs qui auront le droit de pousser (écrire) et le garder pour un accès `git://` en lecture seule. C'est aussi le protocole le plus difficile à mettre en place. Il doit être géré par son propre *daemon* qui est spécifique. Nous traiterons de cette installation dans la section « Gitosis » de ce chapitre — elle nécessite la configuration d'un *daemon* `xinetd` ou apparenté, ce qui est loin d'être simple. Il nécessite aussi un accès à travers le pare-feu au port 9418 qui n'est pas un port ouvert en standard dans les pare-feux professionnels. Derrière les gros pare-feux professionnels, ce port obscur est tout simplement bloqué.

4.1.4 Protocole HTTP/S

Enfin, il reste le protocole HTTP. La beauté d'HTTP ou HTTPS tient dans la simplicité à le mettre en place. Tout ce qu'il y a à faire, c'est de simplement copier un dépôt Git nu sous votre racine de document HTTP et de paramétrer un crochet `post-update` et c'est prêt (voir chapitre 7 pour les détails sur les crochets de Git). À partir de ceci, toute personne possédant un accès au serveur web sur lequel vous avez copié votre dépôt peut le cloner. Pour autoriser un accès en lecture à votre dépôt sur HTTP, faîtes ceci :

```
$ cd /var/www/htdocs/
$ git clone --bare /chemin/vers/git_projet projetgit.git
$ cd projetgit.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

C'est tout. Le crochet `post-update` qui est livré avec Git par défaut lance la commande appropriée (`git update-server-info`) pour permettre un fonctionnement correct du clonage et de la récupération par HTTP. Cette commande est lancée lorsque vous poussez vers ce dépôt par SSH ; ainsi, les autres personnes peuvent cloner via la commande

```
$ git clone http://exemple.com/projetgit.git
```

Dans ce cas particulier, nous utilisons le chemin `/var/www/htdocs` qui est commun pour les installations d'Apache, mais vous pouvez utiliser n'importe quel serveur web de pages statiques — il suffit de placer le dépôt nu dans le chemin d'accès. Les données Git sont servies comme des simples fichiers statiques (voir chapitre 9 pour la manière détaillée dont ils sont servis).

Il est possible de faire pousser Git à travers HTTP, bien que cette technique ne soit pas utilisée et nécessite de gérer les exigences complexes de WebDAV. Comme elle est rarement utilisée, nous ne la détaillerons pas dans ce livre. Si vous êtes tout de même intéressé par l'utilisation des protocoles de push-HTTP, vous pouvez vous référer à http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt. Un des intérêts à permettre de pousser par HTTP est que vous pouvez utiliser sur n'importe quel serveur WebDAV, sans liaison avec Git. Il est donc possible d'utiliser cette fonctionnalité si votre fournisseur d'hébergement web supporte WebDAV pour la mise à jour de vos sites.

Avantages

L'avantage d'utiliser le protocole HTTP est qu'il est très simple à mettre en œuvre. Donner un accès public en lecture à votre dépôt Git ne nécessite que quelques commandes. Cela ne prend que quelques minutes. De plus, le protocole HTTP n'est pas très demandeur en ressources système. Les besoins étant limités à servir des données statiques, un serveur Apache standard peut servir des milliers de fichiers par seconde en moyenne et il est très difficile de surcharger même un ordinateur moyen.

Vous pouvez aussi publier votre dépôt par HTTPS, ce qui signifie que vous pouvez chiffrer le contenu transféré. Vous pouvez même obliger les clients à utiliser des certificats SSL spécifiques. Généralement, si vous souhaitez pousser jusque là, il est préférable de passer par des clés SSH publiques. Cependant, certains cas nécessitent l'utilisation de certificats SSL signés ou d'autres méthodes d'authentification basées sur HTTP pour les accès en lecture seule sur HTTPS.

Un autre avantage indéniable de HTTP est que c'est un protocole si commun que les pare-feux d'entreprise sont souvent paramétrés pour le laisser passer.

Inconvénients

L'inconvénient majeur de servir votre dépôt sur HTTP est que c'est relativement inefficace pour le client. Cela prend généralement beaucoup plus longtemps de cloner ou tirer depuis le dépôt et il en résulte un plus grand trafic réseau et de plus gros volumes de transfert que pour les autres protocoles. Le protocole HTTP est souvent appelé le protocole *idiot* parce qu'il n'a pas l'intelligence de sélectionner seulement les données nécessaires à transférer du fait du manque de traitement dynamique côté serveur. Pour plus d'information sur les différences d'efficacité entre le protocole HTTP et les autres, référez-vous au chapitre 9.

4.2 Installation de Git sur un serveur

Pour réaliser l'installation initiale d'un serveur Git, il faut exporter une dépôt existant dans un nouveau dépôt nu — un dépôt qui ne contient pas de copie de répertoire de travail. C'est généralement simple à faire. Pour cloner votre dépôt en créant un nouveau dépôt nu, lancez la commande clone avec l'option `--bare`. Par convention, les répertoires de dépôt nu finissent en `.git`, de cette manière :

```
$ git clone --bare mon_project mon_project.git
Initialized empty Git repository in /opt/projets/mon_project.git/
```

La sortie de cette commande est un peu déroutante. Comme `clone` est un `git init` de base, suivi d'un `git fetch`, nous voyons les messages du `git init` qui crée un répertoire vide. Le transfert effectif d'objet ne fournit aucune sortie, mais il a tout de même lieu. Vous devriez maintenant avoir une copie des données de Git dans votre répertoire `mon_project.git`.

C'est grossièrement équivalent à

```
$ cp -Rf mon_project/.git mon_project.git
```

Il y a quelques légères différences dans le fichier de configuration mais pour l'utilisation envisagée, c'est très proche. La commande extrait le répertoire Git sans répertoire de travail et crée un répertoire spécifique pour l'accueillir.

4.2.1 Copie du dépôt nu sur un serveur

À présent que vous avez une copie nue de votre dépôt, il ne reste plus qu'à la placer sur un serveur et à régler les protocoles. Supposons que vous avez mis en place un serveur nommé `git.exemple.com` auquel vous avez accès par SSH et que vous souhaitez stocker vos dépôts Git dans le répertoire `/opt/git`. Vous pouvez mettre en place votre dépôt en copiant le dépôt nu :

```
$ scp -r mon_projet.git utilisateur@git.exemple.com:/opt/git
```

A partir de maintenant, tous les autres utilisateurs disposant d'un accès SSH au serveur et ayant un accès en lecture seule au répertoire `/opt/git` peuvent cloner votre dépôt en lançant la commande

```
$ git clone utilisateur@git.exemple.com:/opt/git/mon_projet.git
```

Si un utilisateur se connecte par SSH au serveur et a accès en lecture au répertoire `/opt/git/mon_projet.git`, il aura automatiquement accès pour tirer. Git ajoutera automatiquement les droits de groupe en écriture à un dépôt si vous lancez la commande `git init` avec l'option `--shared`.

```
$ ssh utilisateur@git.exemple.com
$ cd /opt/git/mon_projet.git
$ git init --bare --shared
```

Vous voyez comme il est simple de prendre un dépôt Git, créer une version nue et la placer sur un serveur auquel vous et vos collaborateurs avez accès en SSH. Vous voilà prêts à collaborer sur le même projet.

Il faut noter que c'est littéralement tout ce dont vous avez besoin pour démarrer un serveur Git utile auquel plusieurs personnes ont accès — ajoutez des comptes SSH sur un serveur, et collez un dépôt nu quelque part où tous les utilisateurs ont accès en lecture et écriture. Vous êtes prêts à travailler, vous n'avez besoin de rien d'autre.

Dans les chapitres à venir, nous traiterons de mises en place plus sophistiquées. Ces sujets incluront l'élimination du besoin de créer un compte système pour chaque utilisateur, l'accès public aux dépôts, la mise en place d'interfaces utilisateurs web, l'utilisation de l'outil Gitosis, etc. Néanmoins, gardez à l'esprit que pour collaborer avec quelques personnes sur un projet privé, tout ce qu'il faut, c'est un serveur SSH et un dépôt nu.

4.2.2 Petites installations

Si vous travaillez dans un petit groupe ou si vous n'êtes qu'en phase d'essai de Git au sein de votre société avec peu de développeurs, les choses peuvent rester simples. Un des aspects les plus compliqués de la mise en place d'un serveur Git est la gestion des utilisateurs. Si vous souhaitez que certains dépôts ne soient accessibles à certains utilisateurs qu'en lecture seule et en lecture/écriture pour d'autres, la gestion des accès et des permissions peut devenir difficile à régler.

Accès SSH

Si vous disposez déjà d'un serveur auquel tous vos développeurs ont un accès SSH, il est généralement plus facile d'y mettre en place votre premier dépôt car vous n'aurez quasiment aucun réglage supplémentaire à faire (comme nous l'avons expliqué dans le chapitre précédent). Si vous souhaitez des permissions d'accès plus complexes, vous pouvez les mettre en place par le jeu des permissions standards sur le système de fichier du système d'exploitation de votre serveur.

Si vous souhaitez placer vos dépôts sur un serveur qui ne dispose pas déjà de comptes pour chacun des membres de votre équipe qui aurait accès en écriture, alors vous devrez mettre en place

un accès SSH pour eux. En supposant que pour vos dépôts, vous disposiez déjà d'un serveur SSH installé et sur lequel vous avez accès.

Il y a quelques moyens de donner un accès à tout le monde dans l'équipe. Le premier est de créer des comptes pour tout le monde, ce qui est logique mais peut s'avérer lourd. Vous ne souhaiteriez sûrement pas lancer `adduser` et entrer un mot de passe temporaire pour chaque utilisateur.

Une seconde méthode consiste à créer un seul utilisateur git sur la machine, demander à chaque développeur nécessitant un accès en écriture de vous envoyer une clef publique SSH et d'ajouter la dite clef au fichier `~/.ssh/authorized_keys` de votre utilisateur git. À partir de là, tout le monde sera capable d'accéder à la machine via l'utilisateur git. Cela n'affecte en rien les données de commit — les informations de l'utilisateur SSH par lequel on se connecte n'affectent pas les données de commit enregistrées.

Une dernière méthode consiste à faire une authentification SSH auprès d'un serveur LDAP ou tout autre système d'authentification centralisé que vous utiliseriez déjà. Tant que chaque utilisateur peut accéder à un shell sur la machine, n'importe quel schéma d'authentification SSH devrait fonctionner.

4.3 Génération des clefs publiques SSH

Cela dit, de nombreux serveurs Git utilisent une authentification par clefs publiques SSH. Pour fournir une clef publique, chaque utilisateur de votre système doit la générer s'il n'en a pas déjà. Le processus est similaire sur tous les systèmes d'exploitation. Premièrement, l'utilisateur doit vérifier qu'il n'en a pas déjà une. Par défaut, les clefs SSH d'un utilisateur sont stockées dans le répertoire `~/.ssh` du compte. Vous pouvez facilement vérifier si vous avez déjà une clef en listant le contenu de ce répertoire :

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

Recherchez une paire de fichiers appelés *quelquechose* et *quelquechose*`.pub` où le `quelquechose` en question est généralement `id_dsa` ou `id_rsa`. Le fichier en `.pub` est la clef publique tandis que l'autre est la clef privée. Si vous ne voyez pas ces fichiers (ou n'avez même pas de répertoire `.ssh`), vous pouvez les créer en lançant un programme appelé `ssh-keygen` fourni par le paquet SSH sur les systèmes Linux/Mac et MSysGit pour Windows :

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

Premièrement, le programme demande confirmation pour l'endroit où vous souhaitez sauvegarder la clef (`.ssh/id_rsa`) puis il demande deux fois d'entrer un mot de passe qui peut être laissé vide si vous ne souhaitez pas devoir taper un mot de passe quand vous utilisez la clef.

Maintenant, chaque utilisateur ayant suivi ces indications doit envoyer la clef publique à la personne en charge de l'administration du serveur Git (en supposant que vous utilisez un serveur SSH réglé pour l'utilisation de clefs publiques). Ils doivent copier le contenu du fichier .pub et l'envoyer par e-mail. Les clefs publiques ressemblent à ceci :

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

Pour un tutoriel plus approfondi sur la création de clef SSH sur différents systèmes d'exploitation, référez-vous au guide GitHub sur les clefs SSH à http://github.com/guides/providing-your-ssh-key.

4.4 Mise en place du serveur

Parcourons les étapes de la mise en place d'un accès SSH côté serveur. Dans cet exemple, vous utiliserez la méthode des `authorized_keys` pour authentifier vos utilisateurs. Nous supposerons également que vous utilisez une distribution Linux standard telle qu'Ubuntu. Premièrement, créez un utilisateur 'git' et un répertoire `.ssh` pour cet utilisateur.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Ensuite, vous devez ajouter la clef publique d'un développeur au fichier `authorized_keys` de l'utilisateur git. Supposons que vous avez reçu quelques clefs par e-mail et les avez sauvées dans des fichiers temporaires. Pour rappel, une clef publique ressemble à ceci :

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Il suffit de les ajouter au fichier `authorized_keys` :

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Maintenant, vous pouvez créer un dépôt vide nu en lançant la commande `git init` avec l'option `--bare`, ce qui initialise un dépôt sans répertoire de travail :

```
$ cd /opt/git
$ mkdir projet.git
$ cd projet.git
$ git --bare init
```

Alors, John, Josie ou Jessica peuvent pousser la première version de leur projet vers ce dépôt en l'ajoutant en tant que dépôt distant et en lui poussant une branche. Notons que quelqu'un doit se connecter au serveur et créer un dépôt nu pour chaque ajout de projet. Supposons que le nom du serveur soit `gitserveur`. Si vous l'hébergez en interne et avez réglé le DNS pour faire pointer `gitserver` sur ce serveur, alors vous pouvez utiliser les commandes suivantes telle quelle :

```
# Sur l'ordinateur de John
$ cd monproject
$ git init
$ git add .
$ git commit -m 'premiere validation'
$ git remote add origin git@gitserveur:/opt/git/projet.git
$ git push origin master
```

À présent, les autres utilisateurs peuvent cloner le dépôt et y pousser leurs modifications aussi simplement :

```
$ git clone git@gitserveur:/opt/git/projet.git
$ vim LISEZMOI
$ git commit -am 'correction fichier LISEZMOI'
$ git push origin master
```

De cette manière, vous pouvez rapidement mettre en place un serveur Git en lecture/écriture pour une poignée de développeurs.

En précaution supplémentaire, vous pouvez simplement restreindre l'utilisateur 'git' à des actions Git avec un shell limité appelé `git-shell` qui est fourni avec Git. Si vous positionnez ce shell comme shell de login de l'utilisateur 'git', l'utilisateur git ne peut pas avoir de shell normal sur ce serveur. Pour utiliser cette fonction, spécifiez `git-shell` en lieu et place de bash ou csh pour shell de l'utilisateur. Cela se réalise généralement en éditant le fichier `/etc/passwd` :

```
$ sudo vim /etc/passwd
```

Tout au bas, vous devriez trouver une ligne qui ressemble à ceci :

```
git:x:1000:1000::/home/git:/bin/sh
```

Modifiez `/bin/sh` en `/usr/bin/git-shell` (ou le résultat de la commande `which git-shell` qui indique où il est installé). La ligne devrait maintenant ressembler à ceci :

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

À présent, l'utilisateur 'git' ne peut plus utiliser la connexion SSH que pour pousser et tirer sur des dépôts Git, il ne peut plus ouvrir un shell. Si vous essayez, vous verrez un rejet de login :

```
$ ssh git@gitserveur
fatal: What do you think I am? A shell?
Connection to gitserveur closed.
```

4.5 Accès public

Et si vous voulez permettre des accès anonymes en lecture ? Peut-être souhaitez-vous héberger un projet open source au lieu d'un projet interne privé. Ou peut-être avez-vous quelques serveurs de compilation ou d'intégration continue qui changent souvent et vous ne souhaitez pas avoir à regénérer des clefs SSH tout le temps — vous avez besoin d'un accès en lecture seule simple.

Le moyen le plus simple pour des petites installations est probablement d'installer un serveur web statique dont la racine pointe sur vos dépôts Git puis d'activer le crochet `post-update` mentionné à la première partie de ce chapitre. Reprenons l'exemple précédent. Supposons que vos dépôts soient dans le répertoire `/opt/git` et qu'un serveur Apache soit installé sur la machine. Vous pouvez bien sûr utiliser n'importe quel serveur web mais nous utiliserons Apache pour montrer la configuration nécessaire.

Premièrement, il faut activer le crochet :

```
$ cd projet.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Si vous utilisez une version de Git antérieure à 1.6, la commande `mv` n'est pas nécessaire car Git n'a commencé à utiliser le nommage des exemples de crochet en utilisant le postfixe .sample que récemment.

Quelle est l'action de ce crochet `post-update` ? Il contient simplement ceci :

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

Cela signifie que lorsque vous poussez vers le serveur via SSH, Git lance cette commande pour mettre à jour les fichiers nécessaires lorsqu'on tire par HTTP.

Ensuite, il faut ajouter dans la configuration Apache une entrée VirtualHost dont la racine pointe sur vos dépôts Git. Ici, nous supposerons que vous avez réglé un DNS avec résolution générique qui renvoit `*.gitserveur` vers la machine qui héberge ce système :

```
<VirtualHost *:80>
    ServerName git.gitserveur
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

Vous devrez aussi positionner le groupe d'utilisateur Unix du répertoire `/opt/git` à `www-data` de manière à ce que le serveur web puisse avoir accès en lecture seule aux répertoires si le serveur Apache lance le script CGI avec cet utilisateur (par défaut) :

```
$ chgrp -R www-data /opt/git
```

Après avoir redémarré Apache, vous devriez être capable de cloner vos dépôts en spécifiant l'URL de votre projet :

```
$ git clone http://git.gitserveur/projet.git
```

Ainsi, vous pouvez donner accès en lecture seule à tous vos projets à un grand nombre d'utilisateurs en quelques minutes. Une autre option simple pour fournir un accès public non-authentifié consiste à lancer un daemon Git, bien que cela requiert de daemoniser le processus ⍰ nous traiterons cette option dans un chapitre ultérieur si vous préférez cette option.

4.6 GitWeb

Après avoir réglé les accès de base en lecture/écriture et en lecture seule pour vos projets, vous souhaiterez peut-être mettre en place une interface web simple de visualisation. Git fournit un script CGI appelé GitWeb qui est souvent utilisé à cette fin. Vous pouvez voir GitWeb en action sur des sites tels que `http://git.kernel.org` (voir figure 4-1).

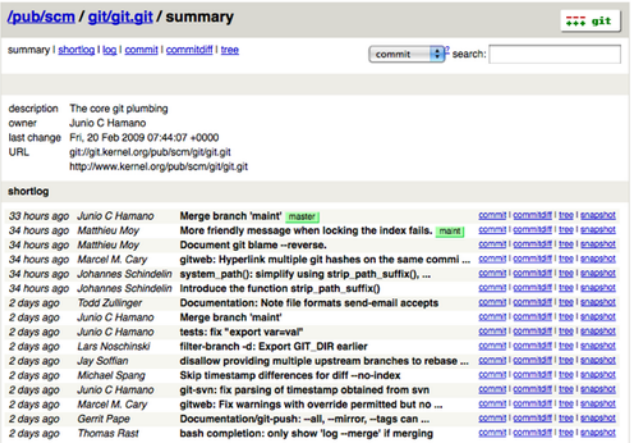

Figure 4.1: *L'interface web de visualisation GitWeb*

Si vous souhaitez vérifier à quoi GitWeb ressemblerait pour votre projet, Git fournit une commande pour démarrer une instance temporaire de serveur si vous avez un serveur léger tel que `lighttpd` ou `webrick` sur votre système. Sur les machines Linux, `lighttpd` est souvent préinstallé et vous devriez pouvoir le démarrer en tapant `git instaweb` dans votre répertoire de travail. Si vous utilisez un Mac, Ruby est installé de base avec Léopard, donc `webrick` est une meilleure option. Pour démarrer `instaweb` avec un gestionnaire autre que lighttpd, vous pouvez le lancer avec l'option `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Cette commande démarre un serveur HTTP sur le port 1234 et lance automatique un navigateur internet qui ouvre la page d'accueil. C'est vraiment très simple. Pour arrêter le serveur, il suffit de lancer la même commande, mais avec l'option `--stop` :

```
$ git instaweb --httpd=webrick --stop
```

Si vous souhaitez fournir l'interface web en permanence sur le serveur pour votre équipe ou pour un projet opensource que vous hébergez, il sera nécessaire d'installer le script CGI pour qu'il soit appelé par votre serveur web. Quelques distributions Linux ont un package `gitweb` qu'il suffira d'installer via `apt` ou `yum`, ce qui est une possibilité. Nous détaillerons tout de même rapidement l'installation manuelle de GitWeb. Premièrement, le code source de Git qui fournit GitWeb est nécessaire pour pouvoir générer un script CGI personnalisé :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
        prefix=/usr gitweb/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

Notez que vous devez indiquer où trouver les dépôts Git au moyen de la variable `GITWEB_PROJECTROOT`. Maintenant, il faut paramétrer dans Apache l'utilisation de CGI pour ce script, en spécifiant un nouveau VirtualHost :

```
<VirtualHost *:80>
    ServerName gitserveur
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Une fois de plus, GitWeb peut être géré par tout serveur Web capable de prendre en charge CGI. La mise en place ne devrait pas être plus difficile avec un autre serveur. Après redémarrage du serveur, vous devriez être capable de visiter `http://gitserveur/` pour visualiser vos dépôts en ligne et de cloner et tirer depuis ces dépôts par HTTP sur `http://git.gitserveur`.

4.7 Gitosis

Conserver les clefs publiques de tous les utilisateurs dans le fichier `authorized_keys` n'est satisfaisant qu'un temps. Avec des centaines d'utilisateurs, la gestion devient compliquée. À chaque fois, il faut se connecter au serveur et il n'y a aucun contrôle d'accès — toute personne avec une clef dans le fichier a accès en lecture et écriture à tous les projets.

Il est temps de se tourner vers un logiciel largement utilisé appelé Gitosis. Gitosis est une collection de scripts qui aident à gérer le fichier `authorized_keys` ainsi qu'à implémenter des contrôles d'accès simples. La partie la plus intéressante de l'outil est que l'interface d'administration permettant d'ajouter des utilisateurs et de déterminer leurs droits n'est pas une interface web mais un dépôt Git spécial. Vous paramétrez les informations dans ce projet et lorsque vous le poussez, Gitosis reconfigure les serveurs en fonction des données, ce qui est cool.

L'installation de Gitosis n'est pas des plus aisées. Elle est plus simple sur un serveur Linux — les exemples qui suivent utilisent une distribution Ubuntu Server 8.10 de base.

Gitosis nécessite des outils Python. Il faut donc installer le paquet Python setuptools qu'Ubuntu fournit en tant que python-setuptools :

```
$ apt-get install python-setuptools
```

Ensuite, il faut cloner et installer Gitosis à partir du site principal du projet :

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
```

```
$ sudo python setup.py install
```

La dernière commande installe deux exécutables que Gitosis utilisera. Ensuite, Gitosis veut gérer ses dépôts sous `/home/git`, ce qui est parfait. Mais vous avez déjà installé vos dépôts sous `/opt/git`, donc au lieu de tout reconfigurer, créez un lien symbolique :

```
$ ln -s /opt/git /home/git/repositories
```

Comme Gitosis gérera vos clefs pour vous, il faut effacer le fichier `authorized_keys`, réajouter les clefs plus tard et laisser Gitosis contrôler le fichier automatiquement. Pour l'instant, déplacez le fichier `authorized_keys` ailleurs :

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Ensuite, il faut réactiver le shell pour l'utilisateur « git » si vous l'avez désactivé au moyen de `git-shell`. Les utilisateurs ne pourront toujours pas se connecter car Gitosis contrôlera cet accès. Modifions la ligne dans le fichier `/etc/passwd`

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

pour la version d'origine :

```
git:x:1000:1000::/home/git:/bin/sh
```

Vous pouvez maintenant initialiser Gitosis en lançant la commande `gitosis-init` avec votre clef publique. Si votre clef publique n'est pas présente sur le serveur, il faut l'y télécharger :

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

Cela permet à l'utilisateur disposant de cette clef de modifier le dépôt Git qui contrôle le paramétrage de Gitosis. Ensuite, il faudra positionner manuellement le bit « execute » du script `post-update` du dépôt de contrôle nouvellement créé.

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

Vous voilà prêt. Si tout est réglé correctement, vous pouvez essayer de vous connecter par SSH au serveur en tant que l'utilisateur pour lequel vous avez ajouté la clef publique lors de l'initialisation de Gitosis.. Vous devriez voir quelque chose comme :

```
$ ssh git@gitserveur
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
  Connection to gitserveur closed.
```

Cela signifie que Gitosis vous a bien reconnu mais vous a rejeté car vous ne lancez pas de commandes Git. Lançons donc une vraie commande Git en clonant le dépôt de contrôle Gitosis :

```
# sur votre ordinateur local
$ git clone git@gitserveur:gitosis-admin.git
```

Vous avez à présent un répertoire `gitosis-admin` qui contient deux entrées :

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

Le fichier `gitosis.conf` est le fichier de configuration qui permet de spécifier les utilisateurs, les dépôts et les permissions. Le répertoire `keydir` stocke les clefs publiques de tous les utilisateurs qui peuvent avoir un accès à vos dépôts — un fichier par utilisateur. Le nom du fichier dans `keydir` (dans l'exemple précédent, `scott.pub`) sera différent pour vous — Gitosis utilise le nom issu de la description à la fin de la clef publique qui a été importée par le script `gitosis-init`.

Le fichier `gitosis.conf` contient la configuration du projet `gitosis-admin` cloné à l'instant :

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

Il indique que l'utilisateur « scott » — l'utilisateur dont la clef publique a servi à initialiser Gitosis — est le seul à avoir accès au projet `gitosis-admin`.

A présent, ajoutons un nouveau projet. Ajoutons une nouvelle section appelée `mobile` où vous listez les développeurs de votre équipe mobile et les projets auxquels ces développeurs ont

accès. Comme « scott » est le seul utilisateur déclaré pour l'instant, vous devrez l'ajouter comme membre unique et vous créerez un nouveau projet appelé `iphone_projet` pour commencer :

```
[group mobile]
writable = iphone_projet
members = scott
```

À chaque modification du projet `gitosis-admin`, il est nécessaire de valider les changements et de les pousser sur le serveur pour qu'ils prennent effet :

```
$ git commit -am 'ajout iphone_projet et groupe mobile'
[master]: created 8962da8: "changed name"
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
   fb27aec..8962da8  master -> master
```

Vous pouvez pousser vers le nouveau `iphone_projet` en ajoutant votre serveur comme dépôt distant dans votre dépôt local de projet et en poussant. Vous n'avez plus besoin de créer manuellement un dépôt nu sur le serveur pour les nouveaux projets. Gitosis les crée automatiquement dès qu'il voit la première poussée :

```
$ git remote add origin git@gitserveur:iphone_projet.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_projet.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

Notez qu'il est inutile de spécifier le chemin distant (en fait, c'est interdit), juste deux points et le nom du projet. Gitosis gère les chemins.

Souhaitant travailler sur ce projet avec vos amis, vous devrez rajouter leurs clefs publics. Plutôt que de les accoler manuellement au fichier `~/.ssh/authorized_keys` de votre serveur, il faut les ajouter, une clef par fichier dans le répertoire `keydir`. Le nom de fichier détermine les noms de utilisateurs dans le fichier `gitosis.conf`. Rajoutons les clefs publiques de John, Josie et Jessica :

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Vous pouvez maintenant les ajouter tous à votre équipe `mobile` pour qu'ils aient accès en lecture/écriture à `iphone_projet` :

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

Après validation et poussée vers le serveur, les quatre utilisateurs sont admis à lire et écrire sur ce projet.

Gitosis fournit aussi des permissions simples. Si vous souhaitez que John n'ait qu'un accès en lecture à ce projet, vous pouvez configurer ceci plutôt :

```
[group mobile]
writable = iphone_projet
members = scott josie jessica

[group mobile_ro]
readonly = iphone_projet
members = john
```

A présent, John peut cloner le projet et récupérer les mises à jour, mais Gitosis lui refusera de pousser sur ce projet. Vous pouvez créer autant que groupes que vous désirez contenant des utilisateurs et projets différents. Vous pouvez aussi spécifier un autre groupe comme membre du groupe (avec le préfixe `@`) pour faire hériter ses membres automatiquement :

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
writable  = iphone_projet
members   = @mobile_committers

[group mobile_2]
writable  = autre_iphone_projet
members   = @mobile_committers john
```

Si vous rencontrez des problèmes, il peut être utile d'ajouter `loglevel=DEBUG` sous la section `[gitosis]`. Si vous avez perdu le droit de pousser en envoyant une configuration vérolée, vous pouvez toujours réparer le fichier `/home/git/.gitosis.conf` sur le serveur — le fichier dans lequel Gitosis lit sa configuration. Pousser sur le projet `gitosis-admin` provoque la recopie du fichier `gitosis.conf` à cet endroit. Si vous éditez ce fichier à la main, il restera dans cet état jusqu'à la prochaine poussée.

4.8 Le daemon Git

Pour garantir les accès publics non authentifiés en lecture à vos projet, il est préférable de dépasser le protocole HTTP et de commencer à utiliser le protocole Git. La raison principale en est la vitesse. Le protocole Git est bien plus efficace et de ce fait plus rapide que le protocole HTTP et fera gagner du temps à vos utilisateurs.

Ce système n'est valable que pour les accès non authentifiés en lecture seule. Si vous mettez ceci en place sur un serveur à l'extérieur de votre pare-feu, il ne devrait être utilisé que pour des projets qui sont destinés à être visibles publiquement par le monde entier. Si le serveur est derrière le pare-feu, il peut être utilisé pour des projets avec accès en lecture seule pour un grand nombre d'utilisateurs ou des ordinateurs (intégration continue ou serveur de compilation) pour lequels vous ne souhaitez pas avoir à gérer des clefs SSH.

En tout cas, le protocole Git est relativement facile à mettre en place. Grossièrement, il suffit de lancer la commande suivante en tant que daemon :

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` autorise le serveur à redémarrer sans devoir attendre que les anciennes connexions expirent, l'option `--base-path` autorise les gens à cloner des projets sans devoir spécifier le chemin complet, et le chemin en fin de ligne indique au daemon Git l'endroit où chercher des dépôts à exporter. Si vous utilisez un pare-feu, il sera nécessaire de rediriger le port 9418 sur la machine hébergeant le serveur.

Transformer ce processus en daemon se réalise par différentes manières qui dépendent du système d'exploitation sur lequel il est lancé. Sur une machine Ubuntu, c'est un script Upstart. Donc dans le fichier

```
/etc/event.d/local-git-daemon
```

le script suivant :

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

Par sécurité, ce daemon devrait être lancé par un utilisateur n'ayant que des droits de lecture seule sur les dépôts — simplement en créant un nouvel utilisateur « git-ro » qui servira à lancer le daemon. Par simplicité, nous le lancerons avec le même utilisateur « git » qui est utilisé par Gitosis.

Au rédémarrage de la machine, votre daemon Git démarrera automatiquement et redémarrera s'il meurt. Pour le lancer sans avoir à redémarrer, vous pouvez lancer ceci :

```
initctl start local-git-daemon
```

Sur d'autres systèmes, le choix reste large, allant de `xinetd` à un script de système `sysvinit` ou à tout autre moyen — tant que le programme est daemonisé et surveillé.

Ensuite, il faut spécifier à votre serveur Gitosis les dépôts à autoriser en accès Git. Si vous ajoutez une section pour chaque dépôt, vous pouvez indiquer ceux que vous souhaitez servir en lecture via votre daemon Git. Par exemple, si vous souhaitez un accès par protocole Git à votre projet iphone, ajoutez ceci à la fin du fichier `gitosis.conf` :

```
[repo iphone_projet]
daemon = yes
```

Une fois cette configuration validée et poussée, votre daemon devrait commencer à servir des requêtes pour ce projet à tout personne ayant accès au port 9518 de votre serveur.

Si vous décidez de ne pas utiliser Gitosis, mais d'utiliser un daemon Git, il faudra lancer les commandes suivantes sur chaque projet que vous souhaitez faire servir par le daemon Git :

```
$ cd /chemin/au/projet.git
$ touch git-daemon-export-ok
```

La présence de ce fichier indique à Git que ce projet peut être servi sans authentification.

Gitosis peut aussi contrôler les projets que GitWeb publie. Premièrement, il faut ajouter au fichier `/etc/gitweb.conf` quelque chose comme :

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Vous pouvez contrôler les projets publiés sur GitWeb en ajoutant ou retirant une propriété `gitweb` au fichier de configuration de Gitosis. Par exemple, si vous voulez que le projet iphone soit visible sur GitWeb, le paramétrage `repo` doit être le suivant :

```
[repo iphone_projet]
daemon = yes
gitweb = yes
```

Maintenant, si vous validez et poussez le projet `gitosis-admin`, GitWeb commencera automatiquement à publier votre projet iphone.

4.9 Git hébergé

Si vous ne vous ne voulez pas vous investir dans la mise en place de votre propre serveur Git, il reste quelques options pour héberger vos projets Git chez un site externe dédié à l'hébergement. Cette méthode offre de nombreux avantages : un site en hébergement est généralement rapide à créer et facilite le démarrage de projets, et n'implique pas de maintenance et de surveillance de serveur. Même si vous montez et faites fonctionner votre serveur en interne, vous souhaiterez surement utiliser un site d'hébergement public pour votre code open source — cela rend généralement plus facile l'accès et l'aide par la communauté.

Aujourd'hui, vous avez à disposition un nombre impressionnant d'options d'hébergement, chacune avec différents avantages et désavantages. Pour une liste à jour, référez-vous à la page GitHosting du wiki principal sur Git :

http://git.or.cz/gitwiki/GitHosting

Comme nous ne pourrons pas les passer toutes en revue, et comme de plus, il s'avère que je travaille pour l'une d'entre elles, nous utiliserons ce chapitre pour détailler la création d'un compte et d'un nouveau projet sur GitHub. Cela vous donnera une idée de ce qui est nécessaire.

GitHub est de loin le plus grand site d'hébergement open source sur Git et c'est aussi un des rares à offrir à la fois des options d'hébergement public et privé, ce qui vous permet de conserver vos codes open source et privés au même endroit. En fait, nous avons utilisé GitHub pour collaborer en privé sur ce livre.

4.9.1 GitHub

GitHub est légèrement différent de la plupart des sites d'hébergement de code dans le sens où il utilise un espace de noms pour les projets. Au lieu d'être principalement orienté projet, GitHub est orienté utilisateur. Cela signifie que lorsque j'héberge mon projet `grit` sur GitHub, vous ne le trouverez pas à `github.com/grit` mais plutôt à `github.com/schacon/grit`. Il n'y a pas de dépôt canonique d'un projet, ce qui permet à un projet de se déplacer d'un utilisateur à l'autre sans transition si le premier auteur abandonne le projet.

GitHub est aussi une société commerciale qui facture les comptes qui utilisent des dépôts privés, mais tout le monde peut rapidement obtenir un compte gratuit pour héberger autant de projets libres que désiré. Nous allons détailler comment faire.

4.9.2 Création d'un compte utilisateur

La première chose à faire, c'est de créer un compte utilisateur gratuit. Visitez la page « Prix et inscription » à `http://github.com/plans` et cliquez sur le bouton « Créer un compte gratuit » de la zone « Gratuit pour l'open source » (voir figure 4-2) qui vous amène à la page d'enregistrement.

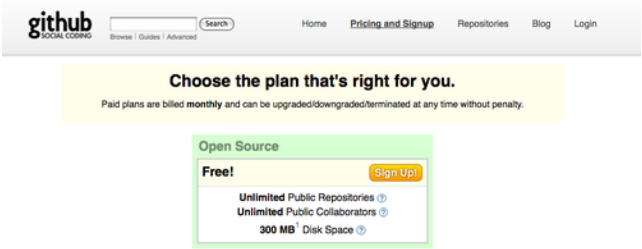

Figure 4.2: *La page des différents plans de GitHub.*

Vous devez choisir un nom d'utilisateur qui n'est pas déjà utilisé dans le système et saisir une adresse e-mail qui sera associée au compte et un mot de passe (voir figure 4-3).

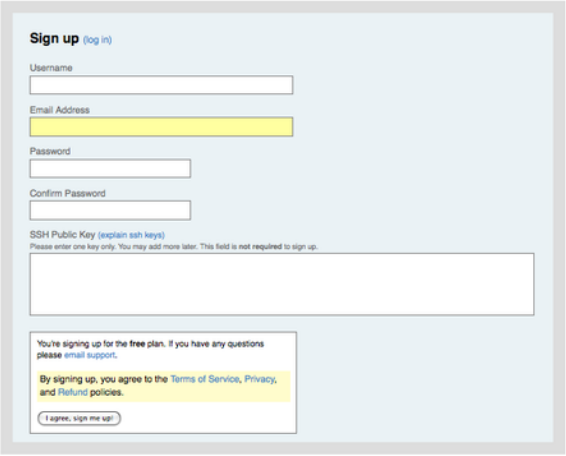

Figure 4.3: *La page d'enregistrement de GitHub*

Si vous l'avez, c'est le bon moment pour ajouter votre clef publique SSH. Nous avons détaillé comment en générer précédemment au chapitre « Petites installations ». Copiez le contenu de la clef publique et collez-le dans la boîte à texte « Clés SSH publiques ». En cliquant sur « Besoin d'aide avec les clés publiques? », vous aurez accès aux instructions (en anglais) pour créer des clefs sur la majorité des systèmes d'exploitation. Cliquez sur « Créer un compte » pour avoir accès au tableau de bord du nouvel utilisateur (voir figure 4-4).

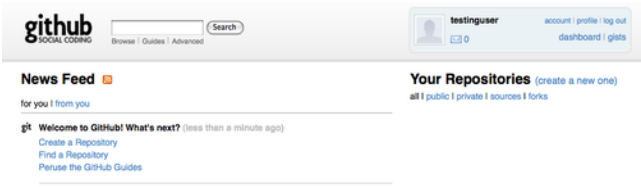

Figure 4.4: *Le tableau de bord d'utilisateur de GitHub*

Vous pouvez ensuite procéder à la création d'un nouveau dépôt.

4.9.3 Création d'un nouveau dépôt

Commencez en cliquant sur « Nouveau dépôt » juste à côté de vos dépôts sur le tableau de bord utilisateur. Un formulaire « Créer un nouveau dépôt » apparaît pour vous guider dans la création d'un nouveau dépôt (voir figure 4-5).

Le strict nécessaire consiste à fournir un nom au projet, mais vous pouvez aussi ajouter une description. Ensuite, cliquez sur le bouton « Créer un dépôt ». Voilà un nouveau dépôt sur GitHub (voir figure 4-6).

Comme il n'y a pas encore de code, GitHub affiche les instructions permettant de créer un nouveau projet, de pousser un projet Git existant ou d'importer un projet depuis un dépôt Subversion public (voir figure 4-7).

Ces instructions sont similaires à ce que nous avons déjà décrit. Pour initialiser un projet qui n'est pas déjà dans Git, tapez

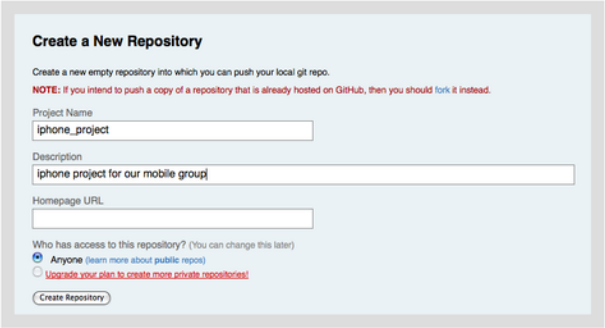

Figure 4.5: *Création d'un nouveau dépôt sur GitHub*

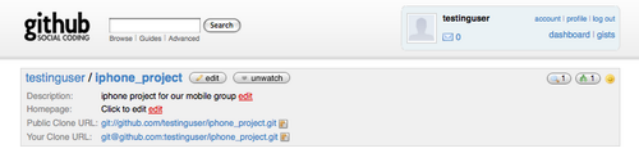

Figure 4.6: *Information principale d'un projet GitHub*

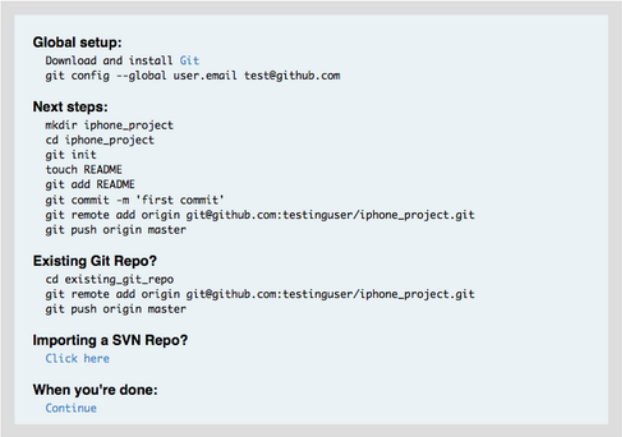

Figure 4.7: *Instructions pour un nouveau dépôt*

```
$ git init
$ git add .
$ git commit -m 'premiere validation'
```

Dans le cas d'un projet Git local, ajoutez GitHub comme dépôt distant et poussez-y votre branche master :

```
$ git remote add origin git@github.com:testinguser/iphone_projet.git
$ git push origin master
```

Votre projet est à présent hébergé sur GitHub et vous pouvez fournir l'URL à toute personne avec qui vous souhaitez le partager. Dans notre cas, il s'agit de `http://github.com/testinguser/`

`iphone_projet`. Vous pouvez aussi voir dans l'entête de la page de chaque projet qu'il y a deux URL Git (voir figure 4-8).

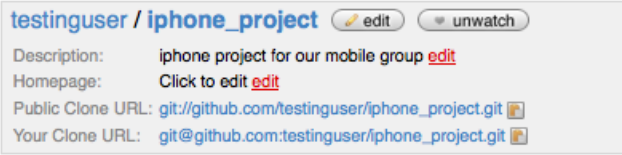

Figure 4.8: *Entête de projet avec une URL publique et une URL privée*

L'URL « Git en lecture seule » est une URL Git publique en lecture seule que tout le monde peut cloner. Utilisez cette URL pour publier et partager votre dépôt sur un site web ou autre.

Votre URL « SSH » est une URL SSH en lecture/écriture qui ne vous permet de lire et écrire que si vous possédez la clef privée associée à la clef publique téléchargée pour votre utilisateur. Quand d'autres utilisateurs visiteront cette page de projet, ils ne verront pas cette URL, ils ne verront que l'URL publique.

4.9.4 Import depuis Subversion

Si vous souhaitez importer un projet public sous Subversion dans Git, GitHub peut vous faciliter la tâche. Il y a un lien « Vous importez un dépôt Subversion? Cliquez ici » au bas de la page d'instructions. En le cliquant, vous accédez à un formulaire contenant des informations sur le processus d'import et une boîte à texte où vous pouvez coller l'URL de votre dépôt public Subversion (voir figure 4-9).

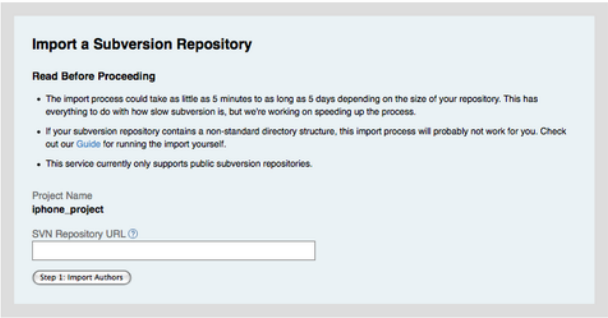

Figure 4.9: *Interface d'import depuis Subversion.*

Si votre projet est très gros, ne suit pas les standards de nommage ou est privé, cette méthone risque de ne pas fonctionner. Au chapitre 7, nous traiterons des imports manuels plus compliqués de projets.

4.9.5 Ajout des collaborateurs

Ajoutons le reste de l'équipe. Si John, Josie et Jessica ouvrent tous un compte sur GitHub, et que vous souhaitez leur donner un accès en écriture à votre dépôt, vous pouvez les ajouter à votre projet comme collaborateurs. Cela leur permettra de pousser leur travail sur le dépôt avec leurs clefs privées.

Cliquez le bouton « Admin » dans l'entête du projet pour accéder à la page d'administration de votre projet GitHub (voir figure 4-10).

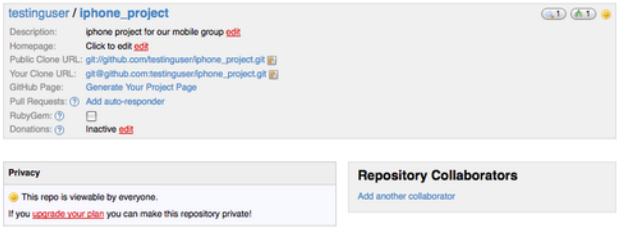

Figure 4.10: *Page d'administration GitHub.*

Pour accorder à un autre utilisateur l'accès en écriture au projet, cliquez l'onglet « Collaborateurs ». Vous pouvez entrer le nom de l'utilisateur dans la boîte à texte qui apparaît. Au fur et à mesure de votre frappe, une liste déroulante affiche les noms qui correspondent aux caractères tapés. Lorsque vous avez trouvé l'utilisateur correct, cliquez le bouton « Ajouter » pour ajouter l'utilisateur comme collaborateur au projet (voir figure 4-11).

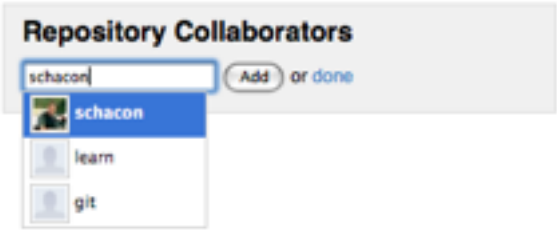

Figure 4.11: *Ajout d'un collaborateur à votre projet.*

Lorsque vous avez fini d'ajouter des collaborateurs, vous devriez les voir en liste dans la boîte « Repository Collaborators » (voir figure 4-12).

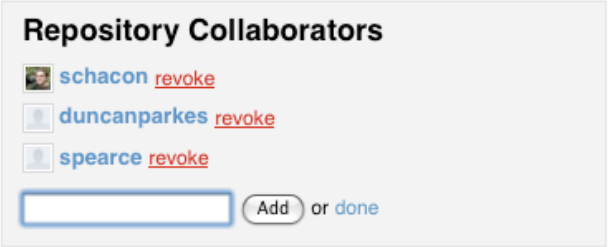

Figure 4.12: *Une liste des collaborateurs sur votre projet.*

Si vous devez révoquer l'accès à certaines personnes, vous pouvez cliquer la croix rouge leur correspondant et leur accès en écriture sera effacé. Pour des projets futurs vous pouvez aussi copier des groupes de collaborateurs en copiant les permissions d'un projet existant.

4.9.6 Votre projet

Une fois que vous avez poussé votre projet ou l'avez importé depuis Subversion, votre page principale de projet ressemble à la figure 4-13.

Lorsqu'on visite votre projet, on voit cette page. Elle contient des onglets vers différentes vues des projets. L'onglet « Contributions » affiche une liste des validations dans l'ordre chronologique

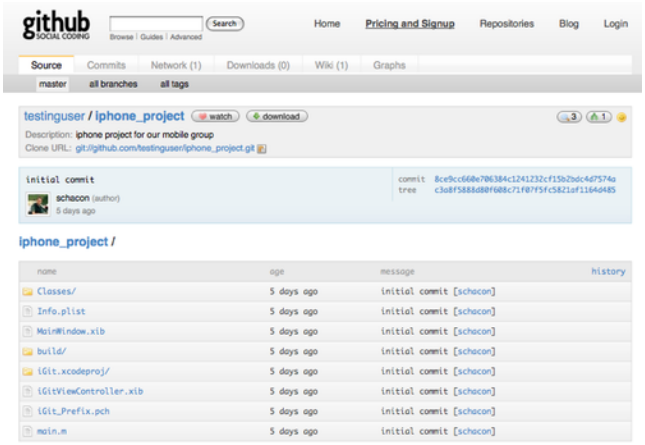

Figure 4.13: *Un page principale de projet GitHub.*

inverse, similaire à ce qu'afficherait la commande `git log`. L'onglet « Réseau » affiche tous les utilisateurs ayant dupliqué votre projet et contribué. L'onglet « Téléchagements » vous permet de télécharger les éxécutables du projet ou de fournir des archives des sources de votre projet à des points balisés. L'onglet « Wiki » fournit un wiki ou vous pouvez commencer à écrire la documentation ou d'autres informations du projet. L'onglet « Graphiques » permet de visualiser les contributions et les statistiques. L'onglet principal « Source » sur lequel vous arrivez par défaut affiche le contenu du répertoire principal du projet et met en forme dessous le fichier README s'il en contient un. Cet onglet affiche aussi un boîte contenant les informations de la dernière validation.

4.9.7 Duplication de projets

Si vous souhaitez contribuer à un projet auquel vous n'avez pas accès en écriture, GitHub encourage à dupliquer le projet. Si le projet vous semble intéressant et que vous souhaitez le modifier, vous pouvez cliquer sur le bouton « Fork » (dupliquer) visible dans l'entête du projet pour faire copier ce projet par GitHub vers votre utilisateur pour que vous puissiez pousser dessus.

De cette manière, les administrateurs de projet n'ont pas à se soucier d'ajouter des utilisateurs comme collaborateurs pour leur donner un accès en écriture. On peut dupliquer un projet et pousser dessus, et le mainteneur principal du projet peut tirer ces modifications en ajoutant les projets dupliqués comme dépôts distants et en fusionnant les changements.

Pour dupliquer un projet, visitez la page du projet (par exemple mojombo/chronic), et cliquez le bouton « Fork » (dupliquer) dans l'entête (voir figure 4-14).

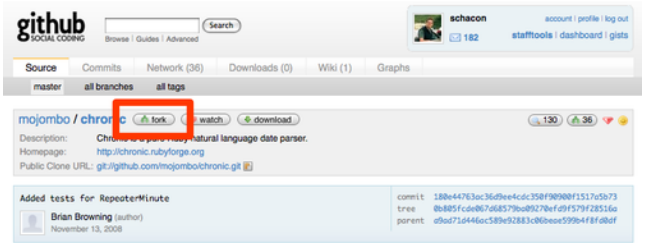

Figure 4.14: *Obtenir un copie modifiable et publiable d'un dépôt en cliquant le bouton « Fork ».*

Quelques secondes plus tard, vous êtes redirigés vers une nouvelle page de projet qui indique que ce projet est un dupliqué d'un autre (voir figure 4-15).

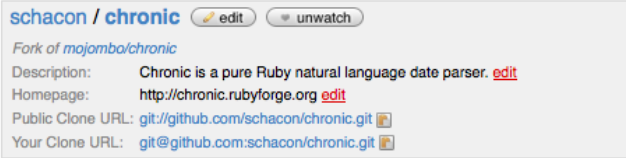

Figure 4.15: *Votre duplicata d'un projet.*

4.9.8 Résumé sur GitHub

C'est tout ce que nous dirons de GitHub, mais il faut souligner que tous ces processus sont très rapides. Vous pouvez créer un compte, ajouter un nouveau projet et commencer à pousser dessus en quelques minutes. Si votre projet est libre, vous pouvez aussi construire une importante communauté de développeurs qui ont à présent la visibilité sur votre projet et peuvent à tout moment le dupliquer et commencer à contribuer. Tout au moins, cela peut s'avérer une manière rapide de démarrer avec Git et de l'essayer.

4.10 Résumé

Vous disposez de plusieurs moyens de mettre en place un dépôt Git distant pour pouvoir collaborer avec d'autres et partager votre travail.

Gérer votre propre serveur vous donne une grande maîtrise et vous permet de l'installer derrière un pare-feu, mais un tel serveur nécessite généralement une certaine quantité de travail pour l'installation et la maintenance. Si vous placez vos données sur un serveur hébergé, c'est très simple à installer et maintenir. Cependant vous devez pouvoir héberger votre code sur des serveurs tiers et certaines politiques d'organisation ne le permettent pas.

Choisir la meilleure solution ou combinaison de solutions pour votre cas ou celui de votre société ne devrait pas poser de problème.

Chapitre 5

Git distribué

Avec un dépôt distant Git mis en place pour permettre à tous les développeurs de partager leur code, et la connaissance des commandes de base de Git pour une gestion locale, abordons les méthodes de gestion distribuée que Git nous offre.

Dans ce chapitre, vous découvrirez comment travailler dans un environnement distribué avec Git en tant que contributeur ou comme intégrateur. Cela recouvre la manière de contribuer efficacement à un projet et de rendre la vie plus facile au mainteneur du projet ainsi qu'à vous-même, mais aussi en tant que mainteneur, de gérer un projet avec de nombreux contributeurs.

5.1 Développements distribués

À la différence des systèmes de gestion de version centralisés (CVCS), la nature distribuée de Git permet une bien plus grande flexibilité dans la manière dont les développeurs collaborent sur un projet. Dans les systèmes centralisés, tout développeur est un nœud travaillant de manière plus ou moins égale sur un concentrateur central. Dans Git par contre, tout développeur est potentiellement un nœud et un concentrateur, c'est-à-dire que chaque développeur peut à la fois contribuer du code vers les autres dépôts et maintenir un dépôt public sur lequel d'autres vont baser leur travail et auquel ils vont contribuer. Cette capacité ouvre une perspective de modes de développement pour votre projet ou votre équipe dont certains archétypes tirant parti de cette flexibilité seront traités dans les sections qui suivent. Les avantages et inconvénients éventuels de chaque mode seront traités. Vous pouvez choisir d'en utiliser un seul ou de mélanger les fonctions de chacun.

5.1.1 Gestion centralisée

Dans les systèmes centralisés, il n'y a généralement qu'un seul modèle de collaboration, la gestion centralisée. Un concentrateur ou dépôt central accepte le code et tout le monde doit synchroniser son travail avec. Les développeurs sont des nœuds, des consommateurs du concentrateur, seul endroit où ils se synchronisent (voir figure 5-1).

Cela signifie que si deux développeurs clonent depuis le concentrateur et qu'ils introduisent tous les deux des modifications, le premier à pousser ses modifications le fera sans encombre. Le second développeur doit fusionner les modifications du premier dans son dépôt local avant de pousser ses modifications pour ne pas écraser les modifications du premier. Ce concept reste aussi vrai avec Git qu'il l'est avec Subversion (ou tout autre CVCS) et le modèle fonctionne parfaitement dans Git.

Si votre équipe est petite et que vous êtes déjà habitués à une gestion centralisée dans votre

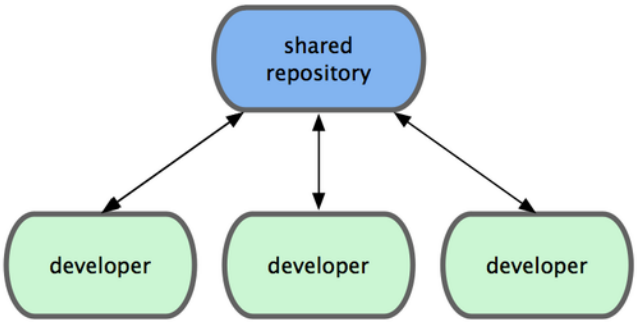

Figure 5.1: *La gestion centralisée.*

société ou votre équipe, vous pouvez simplement continuer à utiliser cette méthode avec Git. Mettez en place un dépôt unique et donnez à tous l'accès en poussée. Git empêchera les utilisateurs d'écraser le travail des autres. Si un développeur clone le dépôt central, fait des modifications et essaie de les pousser alors qu'un autre développeur à poussé ses modifications dans le même temps, le serveur rejettera les modifications du premier. Il lui sera indiqué qu'il cherche à pousser des modifications sans mode avance rapide et qu'il ne pourra pas le faire tant qu'il n'aura pas récupéré et fusionné les nouvelles modifications depuis le serveur. Cette méthode est très intéressante pour de nombreuses personnes car c'est un paradigme avec lequel beaucoup sont familiers et à l'aise.

5.1.2 Mode du gestionnaire d'intégration

Comme Git permet une multiplicité de dépôt distants, il est possible d'envisager un mode de fonctionnement où chaque développeur a un accès en écriture à son propre dépôt public et en lecture à tous ceux des autres. Ce scénario inclut souvent un dépôt canonique qui représente le projet « officiel ». Pour commencer à contribuer au projet, vous créez votre propre clone public du projet et poussez vos modifications dessus. Après, il suffit d'envoyer une demande au mainteneur de projet pour qu'il tire vos modifications dans le dépôt canonique. Il peut ajouter votre dépôt comme dépôt distant, tester vos modifications localement, les fusionner dans sa branche et les pousser vers le dépôt public. Le processus se passe comme ceci (voir figure 5-2) :

1. Le mainteneur du projet pousse vers son dépôt public.
2. Un contributeur clone ce dépôt et introduit des modifications.
3. Le contributeur pousse son travail sur son dépôt public.
4. Le contributeur envoie au mainteneur un e-mail de demande pour tirer depuis son dépôt.
5. Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne localement.
6. Le mainteneur pousse les modifications fusionnées sur le dépôt principal.

C'est une gestion très commune sur des sites tels que GitHub où il est aisé de dupliquer un projet et de pousser ses modifications pour les rendre publiques. Un avantage distinctif de cette approche est qu'il devient possible de continuer à travailler et que le mainteneur du dépôt principal peut tirer les modifications à tout moment. Les contributeurs n'ont pas à attendre le bon-vouloir du mainteneur pour incorporer leurs modifications. Chaque acteur peut travailler à son rythme.

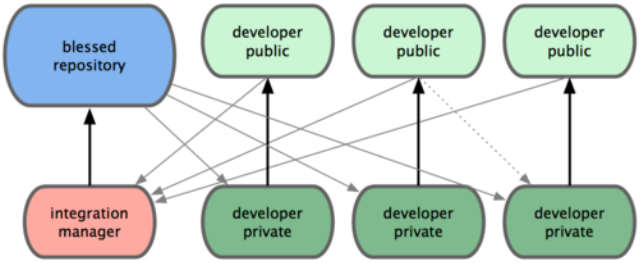

Figure 5.2: *Le mode du gestionnaire d'intégration*

5.1.3 Mode dictateur et ses lieutenants

C'est une variante de la gestion multi-dépôt. En général, ce mode est utilisé sur des projets immenses comprenant des centaines de collaborateurs. Un exemple connu en est le noyau Linux. Des gestionnaires d'intégration gèrent certaines parties du projet. Ce sont les lieutenants. Tous les lieutenants ont un unique gestionnaire d'intégration, le dictateur bénévole. Le dépôt du dictateur sert de dépôt de référence à partir duquel tous les collaborateurs doivent tirer. Le processus se déroule comme suit (voir figure 5-3) :

1. Les développeurs de base travaillent sur la branche thématique et rebasent leur travail sur master. La branche master est celle du dictateur.

2. Les lieutenants fusionnent les branches thématiques des développeurs dans leur propre branche master.

3. Le dictateur fusionne les branches master de ses lieutenants dans sa propre branche master.

4. Le dictateur pousse sa branche master sur le dépôt de référence pour que les développeurs se rebasent dessus.

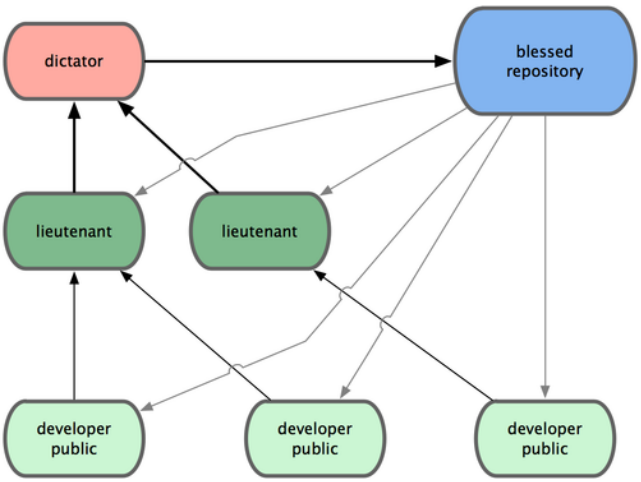

Figure 5.3: *Le processus du dictateur bénévole.*

Ce schéma de processus n'est pas très utilisé mais s'avère utile dans des projets très gros ou pour lesquels un ordre hiérarchique existe, car il permet au chef de projet (le dictateur) de déléguer une grande partie du travail et de collecter de grands sous-ensembles de codes à différents points avant de les intégrer.

Ce sont des schémas de processus rendus possibles et généralement utilisés avec des systèmes distribués tels que Git, mais de nombreuses variations restent possibles pour coller à un flux de modifications donné. En espérant vous avoir aidé à choisir le meilleur mode de gestion pour votre cas, je vais traiter des exemples plus spécifiques de méthode de réalisation des rôles principaux constituant les différents flux.

5.2 Contribution à un projet

Vous savez ce que sont les différents modes de gestion et vous devriez connaître suffisamment l'utilisation de Git. Dans cette section, vous apprendrez les moyens les plus utilisés pour contribuer à un projet.

La principale difficulté à décrire ce processus réside dans l'extraordinaire quantité de variations dans sa réalisation. Comme Git est très flexible, les gens peuvent collaborer de différentes façons et ils le font, et il devient problématique de décrire de manière unique comment devrait se réaliser la contribution à un projet. Chaque projet est légèrement différent. Les variables incluent la taille du corps des contributeurs, le choix du flux de gestion, les accès en validation et la méthode de contribution externe.

La première variable est la taille du corps de contributeurs. Combien de personnes contribuent activement du code sur ce projet et à quelle vitesse ? Dans de nombreux cas, vous aurez deux à trois développeurs avec quelques validations par jour, voire moins pour des projets endormis. Pour des sociétés ou des projets particulièrement grands, le nombre de développeurs peut chiffrer à des milliers, avec des dizaines, voire des centaines de patchs ajoutés chaque jour. Ce cas est important car avec de plus en plus de développeurs, les problèmes de fusion et d'application de patch deviennent de plus en plus courants. Les modifications soumises par un développeur peuvent être rendu obsolètes ou impossibles à appliquer sur des changements qui ont eu lieu dans l'intervalle de leur développement, de leur approbation ou de leur application. Comment dans ces conditions conserver son code en permanence synchronisé et ses patchs valides ?

La variable suivante est le mode de gestion utilisé pour le projet. Est-il centralisé avec chaque développeur ayant un accès égal en écriture sur la ligne de développement principale ? Le projet présente-t-il un mainteneur ou un gestionnaire d'intégration qui vérifie tous les patchs ? Tous les patchs doivent-ils subir une revue de pair et une approbation ? Faîtes-vous partie du processus ? Un système à lieutenant est-il en place et doit-on leur soumettre les modifications en premier ?

La variable suivante est la gestion des accès en écriture. Le mode de gestion nécessaire à la contribution au projet est très différent selon que vous ayez ou non accès au dépôt en écriture. Si vous n'avez pas accès en écriture, quelle est la méthode préférée pour la soumission de modifications ? Y a-t-il seulement un politique en place ? Quelle est la quantité de modifications fournie à chaque fois ? Quelle est la périodicité de contribution ?

Toutes ces questions affectent la manière de contribuer efficacement à un projet et les modes de gestion disponibles ou préférables. Je vais traiter ces sujets dans une série de cas d'utilisation allant des plus simples aux plus complexes. Vous devriez pouvoir construire vos propres modes de gestion à partir de ces exemples.

5.2.1 Guides pour une validation

Avant de passer en revue les cas d'utilisation spécifiques, voici un point rapide sur les messages de validation. La définition et l'utilisation d'un bonne ligne de conduite sur les messages de valida-

tion facilitent grandement l'utilisation de Git et la collaboration entre développeurs. Le projet Git fournit un document qui décrit un certain nombre de bonnes pratiques pour créer des commits qui serviront à fournir des patchs — le document est accessibles dans les sources de Git, dans le fichier `Documentation/SubmittingPatches`.

Premièrement, il ne faut pas soumettre de patchs comportant des erreurs d'espace (caractères espace inutiles en fin de ligne). Git fournit un moyen simple de le vérifier — avant de valider, lancez la commande `git diff --check` qui identifiera et listera les erreurs d'espace. Voici un exemple dans lequel les caractères en couleur rouge ont été remplacés par des `X` :

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+    def command(git_cmd)XXXX
```

En lançant cette commande avant chaque validation, vous pouvez vérifier que vous ne commettez pas d'erreurs d'espace qui pourraient ennuyer les autres développeurs.

Ensuite, assurez-vous de faire de chaque validation une modification logiquement atomique. Si possible, rendez chaque modification digeste — ne codez pas pendant un week-end entier sur cinq sujets différents pour enfin les soumettre tous dans une énorme validation le lundi suivant. Même si vous ne validez pas du week-end, utilisez la zone d'index le lundi pour découper votre travail en au moins une validation par problème, avec un message utile par validation. Si certaines modifications touchent au même fichier, essayez d'utiliser `git add --patch` pour indexer partiellement des fichiers (cette fonctionnalité est traitée au chapitre 6). L'instantané final sera identique, que vous utilisiez une validation unique ou cinq petites validations, à condition que toutes les modifications soient intégrées à un moment, donc n'hésitez pas à rendre la vie plus simple à vos compagnons développeurs lorsqu'ils auront à vérifier vos modifications. Cette approche simplifie aussi le retrait ou l'inversion ultérieurs d'une modification en cas de besoin. Le chapitre 6 décrit justement quelques trucs et astuces de Git pour réécrire l'historique et indexer interactivement les fichiers — utilisez ces outils pour fabriquer un historique propre et compréhensible.

Le dernier point à soigner est le message de validation. S'habituer à écrire des messages de validation de qualité facilite grandement l'emploi et la collaboration avec Git. En règle générale, les messages doivent débuter par une ligne unique d'au plus 50 caractères décrivant concisément la modification, suivie d'un ligne vide, suivie d'une explication plus détaillée. Le projet Git exige que l'explication détaillée inclut la motivation de la modification en contrastant le nouveau comportement par rapport à l'ancien — c'est une bonne règle de rédaction. Un bonne règle consiste aussi à utiliser le présent de l'impératif ou des verbes substantivés dans le message. En d'autres termes, utilisez des ordres. Au lieu d'écrire « J'ai ajouté des tests pour » ou « En train d'ajouter des tests pour », utilisez juste « Ajoute des tests pour » ou « Ajout de tests pour ».

Voici ci-dessous un modèle tiré de celui écrit par Tim Pope at tpope.net :

```
Court résumé des modifications( 50 caractères ou moins)
```

```
Explication plus détaillée si nécessaire. Retour à la ligne vers 72
    caractères. Dans certains contextes, la première ligne est traitée
    comme le sujet d'un e-mail et le reste comme le corps. La ligne
    vide qui sépare le titre du corps est importante (à moins d'omettre
    totalement le corps). Des outils tels que rebase peuvent être gênés
    si vous les laissez collés.

    Paragraphes supplémentaires après des lignes vides.

     - les listes à puce sont acceptées aussi

 - Typiquement, un tiret ou un astérisk précédés d'un espace unique

 unique, séparés par des lignes vides mais les conventions peuvent
varier
```

Si tous vos messages de validation ressemblent à ceci, les choses seront beaucoup plus simples pour vous et les développeurs avec qui vous travaillez. Le projet Git montre des messages de commit bien formatés — je vous encourage à y lancer un `git log --no-merges` pour pouvoir voir comment rend un historique de messages bien formatés.

Dans les exemples suivants et à travers tout ce livre, par soucis de simplification, je ne formaterai pas les messages aussi proprement. J'utiliserai plutôt l'option `-m` de `git commit`. Faites ce que je dis, pas ce que je fais.

5.2.2 Cas d'une petite équipe privée

Le cas le plus probable que vous rencontrerez est celui du projet privé avec un ou deux autres développeurs. Par privé, j'entends source fermé non accessible au public en lecture. Vous et les autres développeurs aurez accès en poussée au dépôt.

Dans cet environnement, vous pouvez suivre une méthode similaire à ce que vous feriez en utilisant Subversion ou tout autre système centralisé. Vous bénéficiez toujours d'avantages tels que la validation hors-ligne et la gestion de branche et de fusion grandement simplifiée mais les étapes restent similaires. La différence principale reste que les fusions ont lieu du côté client plutôt que sur le serveur au moment de valider. Voyons à quoi pourrait ressembler la collaboration de deux développeurs sur un dépôt partagé. Le premier développeur, John, clone le dépôt, fait une modification et valide localement. Dans les exemples qui suivent, les messages de protocole sont remplacés par `...` pour les raccourcir .

```
# Ordinateur de John
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Eliminer une valeur par defaut invalide'
[master 738ee87] Eliminer une valeur par defaut invalide
 1 files changed, 1 insertions(+), 1 deletions(-)
```

La deuxième développeuse, Jessica, fait la même chose. Elle clone le dépôt et valide une modification :

```
# Ordinateur de Jessica
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Ajouter une tache reset'
[master fbff5bc] Ajouter une tache reset
 1 files changed, 1 insertions(+), 0 deletions(-)
```

À présent, Jessica pousse son travail sur le serveur :

```
# Ordinateur de Jessica
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John tente aussi de pousser ses modifications :

```
# Ordinateur de John
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

John n'a pas le droit de pousser parce que Jessica a déjà poussé dans l'intervalle. Il est très important de comprendre ceci si vous avez déjà utilisé Subversion, parce qu'il faut remarquer que les deux développeurs n'ont pas modifié le même fichier. Quand des fichiers différents ont été modifiés, Subversion réalise cette fusion automatiquement sur le serveur alors que Git nécessite une fusion des modifications locale. John doit récupérer les modifications de Jessica et les fusionner avant d'être autorisé à pousser :

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

À présent, le dépôt local de John ressemble à la figure 5-4.

John a une référence aux modifications que Jessica a poussées, mais il doit les fusionner dans sa propre branche avant de pouvoir pousser :

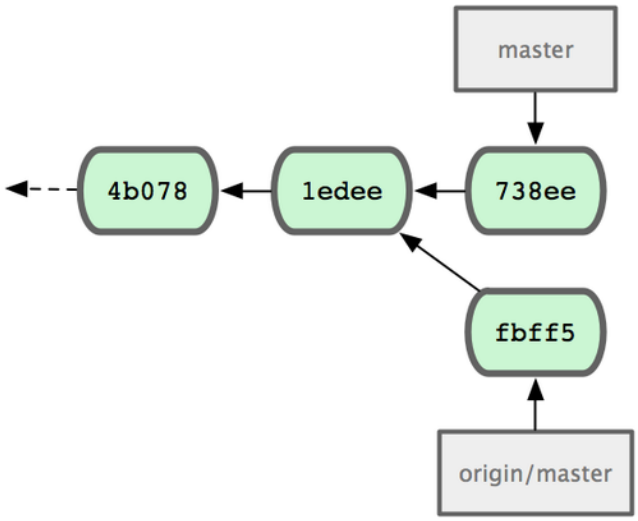

Figure 5.4: *État initial du dépôt de John.*

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Cette fusion se passe sans problème — l'historique de commits de John ressemble à présent à la figure 5-5.

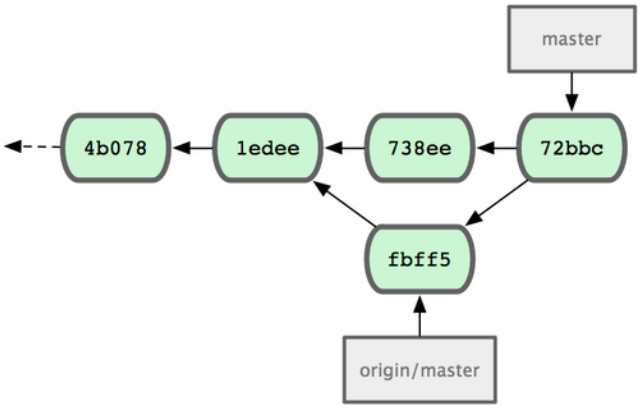

Figure 5.5: *Le dépôt local de John après la fusion d'origin/master.*

Maintenant, John peut tester son code pour s'assurer qu'il fonctionne encore correctement et peut pousser son travail nouvellement fusionné sur le serveur :

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

À la fin, l'historique des commits de John ressemble à la figure 5-6.

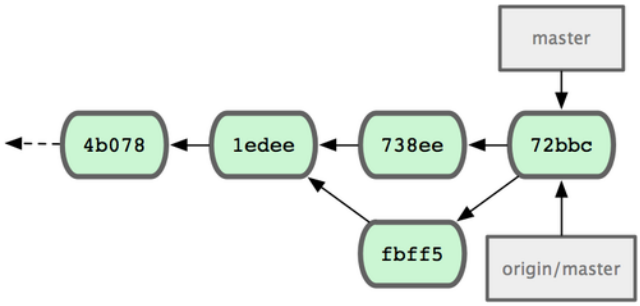

Figure 5.6: *L'historique de John après avoir poussé sur le serveur origin.*

Dans l'intervalle, Jessica a travaillé sur une branche thématique. Elle a créé une branche thématique nommée `prob54` et réalisé trois validations sur cette branche. Elle n'a pas encore récupéré les modifications de John, ce qui donne un historique semblable à la figure 5-7.

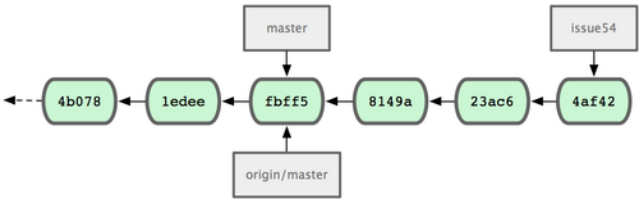

Figure 5.7: *L'historique initial de commits de Jessica.*

Jessica souhaite se synchroniser sur le travail de John. Elle récupère donc ses modifications :

```
# Ordinateur de Jessica
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Cette commande tire le travail que John avait poussé dans l'intervalle. L'historique de Jessica ressemble maintenant à la figure 5-8.

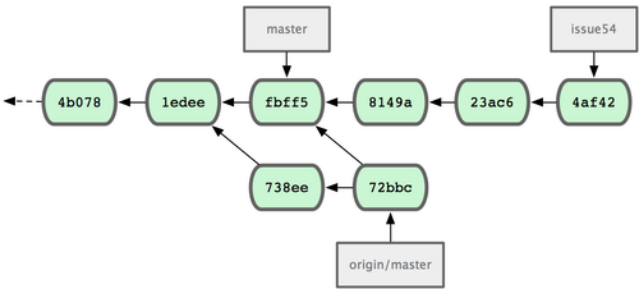

Figure 5.8: *L'historique de Jessica après avoir récupéré les modifications de John.*

Jessica pense que sa branche thématique et prête mais elle souhaite savoir si elle doit fusionner son travail avant de pouvoir pousser. Elle lance `git log` pour s'en assurer :

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    Eliminer une valeur par defaut invalide
```

Maintenant, Jessica peut fusionner sa branche thématique dans sa branche `master`, fusionner le travail de John (`origin/master`)dans sa branche `master`, puis pousser le résultat sur le serveur. Premièrement, elle rebascule sur sa branche `master` pour intégrer son travail :

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Elle peut fusionner soit `origin/master` soit `prob54` en premier — les deux sont en avance, mais l'ordre n'importe pas. L'instantané final devrait être identique quelque soit l'ordre de fusion qu'elle choisit. Seul l'historique sera légèrement différent. Elle choisit de fusionner en premier `prob54` :

```
$ git merge prob54
Updating fbff5bc..4af4298
Fast forward
 LISEZMOI         |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Aucun problème n'apparaît. Comme vous pouvez le voir, c'est une simple avance rapide. Maintenant, Jessica fusionne le travail de John (`origin/master`) :

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Tout a fusionné proprement et l'historique de Jessica ressemble à la figure 5-9.

Maintenant `origin/master` est accessible depuis la branche `master` de Jessica, donc elle devrait être capable de pousser (en considérant que John n'a pas encore poussé dans l'intervalle) :

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

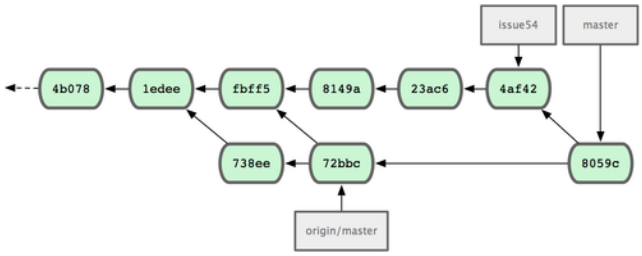

Figure 5.9: *L'historique de Jessica après avoir fusionné les modifications de John.*

Chaque développeur a validé quelques fois et fusionné les travaux de l'autre avec succès (voir figure 5-10).

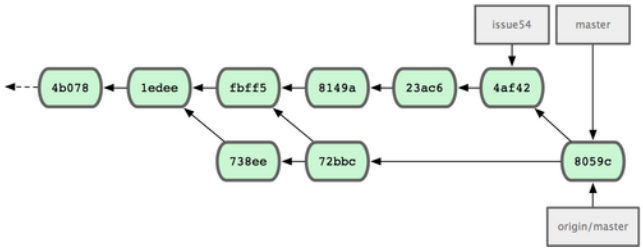

Figure 5.10: *L'historique de Jessica après avoir poussé toutes ses modifications sur le serveur.*

C'est un des schéma les plus simples. Vous travaillez pendant quelques temps, généralement sur une branche thématique, et fusionnez dans votre branche `master` quand elle est prête à être intégrée. Quand vous souhaitez partager votre travail, vous récupérez `origin/master` et la fusionnez si elle a changé, puis finalement vous poussez le résultat sur la branche `master` du serveur. La séquence est illustrée par la figure 5-11.

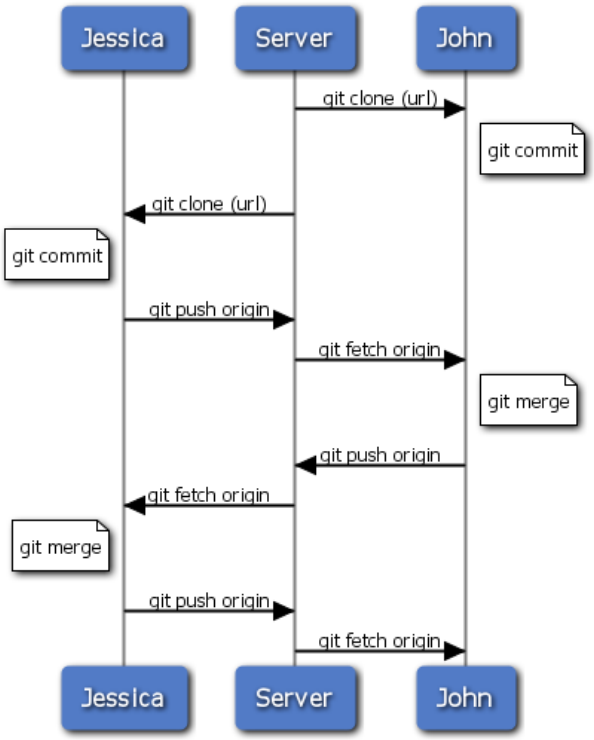

Figure 5.11: *Séquence générale des évènements pour une utilisation simple multi-développeur de Git.*

5.2.3 Équipe privée importante

Dans le scénario suivant, nous aborderons les rôles de contributeur dans un groupe privé plus grand. Vous apprendrez comment travailler dans un environnement où des petits groupes collaborent sur des fonctionnalités, puis les contributions de chaque équipe sont intégrées par une autre entité.

Supposons que John et Jessica travaillent ensemble sur une première fonctionnalité, tandis que Jessica et Josie travaillent sur une autre. Dans ce cas, l'entreprise utilise un mode d'opération de type « gestionnaire d'intégration » où le travail des groupes est intégré par certains ingénieurs, et la branche `master` du dépôt principal ne peut être mise à jour que par ces ingénieurs. Dans ce scénario, tout le travail est validé dans des branches orientées équipe, et tiré plus tard par les intégrateurs.

Suivons le cheminement de Jessica tandis qu'elle travaille sur les deux nouvelles fonctionnalités, collaborant en parallèle avec deux développeurs différents dans cet environnement. En supposant qu'elle a cloné son dépôt, elle décide de travailler sur la `fonctionA` en premier. Elle crée une nouvelle branche pour cette fonction et travaille un peu dessus :

```
# Ordinateur de Jessica
$ git checkout -b fonctionA
Switched to a new branch "fonctionA"
$ vim lib/simplegit.rb
$ git commit -am 'Ajouter une limite à la fonction de log'
[fonctionA 3300904] Ajouter une limite à la fonction de log
 1 files changed, 1 insertions(+), 1 deletions(-)
```

À ce moment, elle a besoin de partager son travail avec John, donc elle pousse les commits de sa branche `fonctionA` sur le serveur. Jessica n'a pas le droit de pousser sur la branche `master` — seuls les intégrateurs l'ont — et elle doit donc pousser sur une autre branche pour collaborer avec John :

```
$ git push origin fonctionA
...
To jessica@githost:simplegit.git
 * [new branch]      fonctionA -> fonctionA
```

Jessica envoie un e-mail à John pour lui indiquer qu'elle a poussé son travail dans la branche appelée `fonctionA` et qu'il peut l'inspecter. Pendant qu'elle attend le retour de John, Jessica décide de commencer à travailler sur la `fonctionB` avec Josie. Pour commencer, elle crée une nouvelle branche thématique, à partir de la base `master` du serveur :

```
# Ordinateur de Jessica
$ git fetch origin
$ git checkout -b fonctionB origin/master
Switched to a new branch "fonctionB"
```

À présent, Jessica réalise quelques validations sur la branche `fonctionB` :

```
$ vim lib/simplegit.rb
$ git commit -am 'Rendre la fonction ls-tree recursive'
[fonctionB e5b0fdc] Rendre la fonction ls-tree recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'Ajout de ls-files'
[fonctionB 8512791] Ajout ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Le dépôt de Jessica ressemble à la figure 5-12.

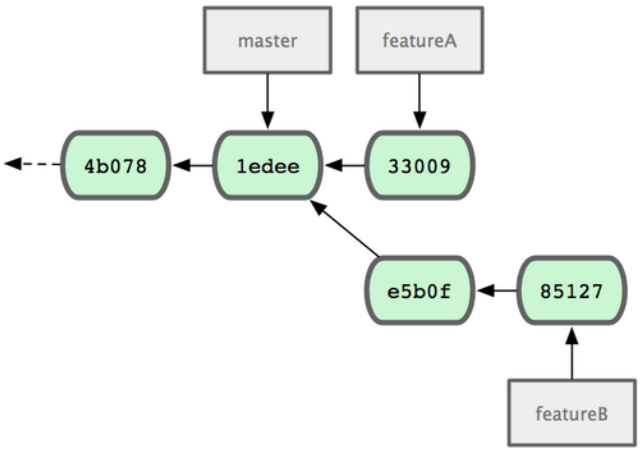

Figure 5.12: *L'historique initial de Jessica.*

Elle est prête à pousser son travail, mais elle reçoit un mail de Josie indiquant qu'une branche avec un premier travail a déjà été poussé sur le serveur en tant que `fonctionBee`. Jessica doit d'abord fusionner ces modifications avec les siennes avant de pouvoir pousser sur le serveur. Elle peut récupérer les modifications de Josie avec `git fetch` :

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      fonctionBee -> origin/fonctionBee
```

Jessica peut à présent fusionner ceci dans le travail qu'elle a réalisé grâce à `git merge` :

```
$ git merge origin/fonctionBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Mais il y a un petit problème — elle doit pousser son travail fusionné dans sa branche `fonctionB` sur la branche `fonctionBee` du serveur. Elle peut le faire en spécifiant la branche locale suivie de deux points (:) suivi de la branche distante à la commande `git push` :

```
$ git push origin fonctionB:fonctionBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  fonctionB -> fonctionBee
```

Cela s'appelle une *refspec*. Référez-vous au chapitre 9 pour une explication plus détaillée des refspecs Git et des possibilités qu'elles offrent.

Ensuite, John envoie un e-mail à Jessica pour lui indiquer qu'il a poussé des modifications sur la branche `fonctionA` et lui demander de les vérifier. Elle lance `git fetch` pour tirer toutes ces modifications :

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  fonctionA   -> origin/fonctionA
```

Elle peut alors voir ce qui a été modifié avec `git log` :

```
$ git log origin/fonctionA ^fonctionA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    largeur du log passee de 25 a 30
```

Finalement, elle fusionne le travail de John dans sa propre branche `fonctionA` :

```
$ git checkout fonctionA
Switched to branch "fonctionA"
$ git merge origin/fonctionA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica veut régler quelques détails. Elle valide donc encore et pousse ses changements sur le serveur :

```
$ git commit -am 'details regles'
[fonctionA ed774b3] details regles
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin fonctionA
...
To jessica@githost:simplegit.git
   3300904..ed774b3  fonctionA -> fonctionA
```

L'historique des commits de Jessica ressemble à présent à la figure 5-13.

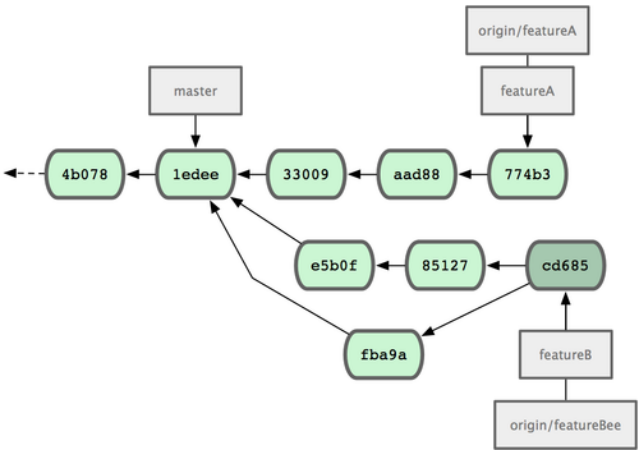

Figure 5.13: *L'historique de Jessica après la validation dans le branche thématique.*

Jessica, Josie et John informent les intégrateurs que les branches `fonctionA` et `fonctionB` du serveur sont prêtes pour une intégration dans la branche principale. Après cette intégration, une synchronisation apportera les commits de fusion, ce qui donnera un historique comme celui de la figure 5-14.

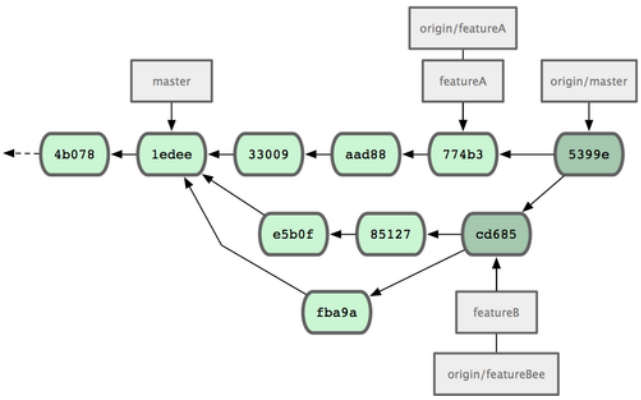

Figure 5.14: *L'historique de Jessica après la fusion de ses deux branches thématiques.*

De nombreuses équipes basculent vers Git du fait de cette capacité à gérer plusieurs équipes travaillant en parallèle, fusionnant plusieurs lignes de développement très tard dans le processus de livraison. La capacité donnée à plusieurs sous-groupes d'équipes à collaborer au moyen de branches distantes sans nécessairement impacter le reste de l'équipe est un grand bénéfice apporté par Git. La séquence de travail qui vous a été décrite ressemble à la figure 5-15.

5.2.4 Petit projet public

Contribuer à un projet public est assez différent. Il faut présenter le travail au mainteneur d'une autre manière parce que vous n'avez pas possibilité de mettre à jour directement des branches du projet. Ce premier exemple décrit un mode de contribution via des serveurs Git qui proposent facilement la duplication de dépôt. Les site repo.or.cz ou GitHub proposent cette méthode, et de nombreux mainteneurs s'attendent à ce style de contribution. Le chapitre suivant traite des projets qui préfèrent accepter les contributions sous forme de patch via e-mail.

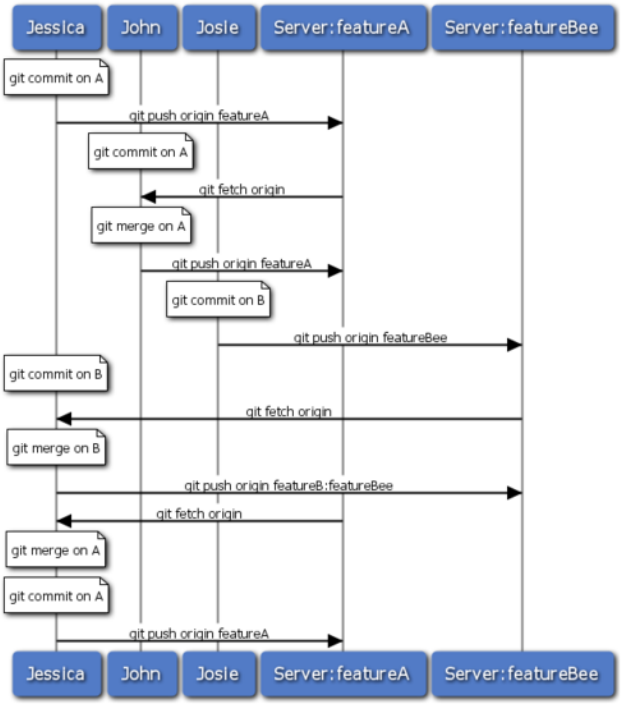

Figure 5.15: *Une séquence simple de gestion orientée équipe.*

Premièrement, vous souhaiterez probablement cloner le dépôt principal, créer une nouvelle branche thématique pour le patch ou la série de patchs que seront votre contribution et commencer à travailler. La séquence ressemble globalement à ceci :

```
$ git clone (url)
$ cd projet
$ git checkout -b fonctionA
$ (travail)
$ git commit
$ (travail)
$ git commit
```

Vous pouvez utiliser `rebase -i` pour réduire votre travail à une seule validation ou pour réarranger les modifications dans des commits qui rendront les patchs plus faciles à relire pour le mainteneur — référez-vous au chapitre 6 pour plus d'information sur comment rebaser de manière interactive.

Lorsque votre branche de travail est prête et que vous êtes prêt à la fournir au mainteneur, rendez-vous sur la page du projet et cliquez sur le bouton « Fork » pour créer votre propre projet dupliqué sur lequel vous aurez les droits en écriture. Vous devez alors ajouter l'URL de ce nouveau dépôt en tant que second dépôt distant, dans notre cas nommé `macopie` :

```
$ git remote add macopie (url)
```

Vous devez pousser votre travail sur cette branche distante. C'est beaucoup plus facile de pousser la branche sur laquelle vous travaillez sur une branche distante que de fusionner et de

poussez le résultat sur le serveur. La raison principale en est que si le travail n'est pas accepté ou s'il est picoré, vous n'aurez pas à faire marche arrière sur votre branche master. Si le mainteneur fusionne, rebase ou picore votre travail, vous le saurez en tirant depuis son dépôt :

```
$ git push macopie fonctionA
```

Une fois votre travail poussé sur votre copie du dépôt, vous devez notifier le mainteneur. Ce processus est souvent appelé une demande de tirage (pull request) et vous pouvez la générer soit via le site web — GitHub propose un bouton « pull request » qui envoie automatiquement un message au mainteneur — soit lancer la commande `git request-pull` et envoyer manuellement par e-mail le résultat au mainteneur de projet.

La commande `request-pull` prend en paramètres la branche de base dans laquelle vous souhaitez que votre branche thématique soit fusionnée et l'URL du dépôt Git depuis lequel vous souhaitez qu'elle soit tirée, et génère un résumé des modifications que vous demandez à faire tirer. Par exemple, si Jessica envoie à John une demande de tirage et qu'elle a fait deux validations dans la branche thématique qu'elle vient de pousser, elle peut lancer ceci :

```
$ git request-pull origin/master macopie
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        ajout d'une nouvelle fonction

are available in the git repository at:

  git://githost/simplegit.git fonctionA

Jessica Smith (2):
      Ajout d'une limite à la fonction de log
      change la largeur du log de 25 a 30

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Le résultat peut être envoyé au mainteneur — cela lui indique d'où la modification a été branchée, le résumé des validations et d'où tirer ce travail.

Pour un projet dont vous n'êtes pas le mainteneur, il est généralement plus aisé de toujours laisser la branche `master` suivre `origin\master` et de réaliser vos travaux sur des branches thématiques que vous pourrez facilement effacer si elles sont rejetées. Garder les thèmes de travaux isolés sur des branches thématiques facilite aussi leur rebasage si le sommet du dépôt principal a avancé dans l'intervalle et que vos modifications ne s'appliquent plus proprement. Par exemple, si vous souhaitez soumettre un second sujet de travail au projet, ne continuez pas à travailler sur la branche thématique que vous venez de pousser mais démarrez en plutôt une depuis la branche `master` du dépôt principal :

```
$ git checkout -b fonctionB origin/master
```

```
$ (travail)
$ git commit
$ git push macopie fonctionB
$ (email au mainteneur)
$ git fetch origin
```

À présent, chaque sujet est contenu dans son propre silo — similaire à un file de patchs — que vous pouvez réécrire, rebaser et modifier sans que les sujets n'interfèrent ou ne dépendent entre eux, comme sur la figure 5-16.

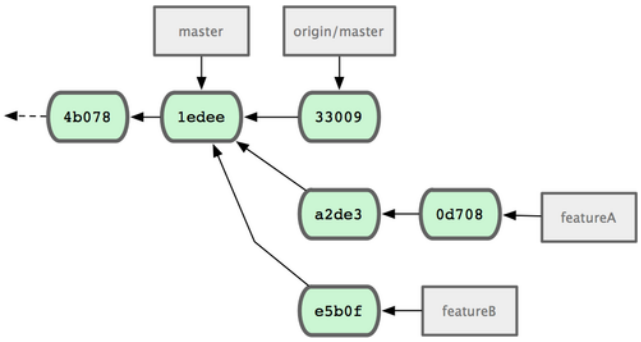

Figure 5.16: *Historique initial des commits avec les modification de fonctionB.*

Supposons que le mainteneur du projet a tiré une poignée d'autres patchs et essayé par la suite votre première branche, mais celle-ci ne s'applique plus proprement. Dans ce cas, vous pouvez rebaser cette branche au sommet de `origin/master`, résoudre les conflits pour le mainteneur et soumettre de nouveau vos modifications :

```
$ git checkout fonctionA
$ git rebase origin/master
$ git push -f macopie fonctionA
```

Cette action réécrit votre historique pour qu'il ressemble à la figure 5-17.

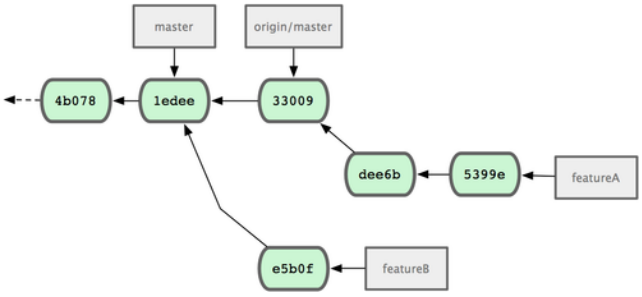

Figure 5.17: *Historique des validations après le travail sur fonctionA.*

Comme vous avez rebasé votre branche, vous devez spécifier l'option `-f` à votre commande pour pousser, pour forcer le remplacement de la branche `fonctionA` sur le serveur par la suite de commits qui n'en est pas descendante. Une solution alternative serait de pousser ce nouveau travail dans une branche différente du serveur (appelée par exemple `fonctionAv2`).

Examinons un autre scénario possible : le mainteneur a revu les modifications dans votre seconde branche et apprécie le concept, mais il souhaiterait que vous changiez des détails d'implémentation. Vous en profiterez pour rebaser ce travail sur le sommet actuel de la branche `master` du projet. Vous démarrez une nouvelle branche à partir de la branche `origin/master` courante, y collez les modifications de `fonctionB` en résolvant les conflits, changez l'implémentation et poussez le tout en tant que nouvelle branche :

```
$ git checkout -b fonctionBv2 origin/master
$ git merge --no-commit --squash fonctionB
$ (changement d'implémentation)
$ git commit
$ git push macopie fonctionBv2
```

L'option `--squash` prend tout le travail de la branche à fusionner et le colle dans un commit sans fusion au sommet de la branche extraite. L'option `--no-commit` indique à Git de ne pas enregistrer automatiquement une validation. Cela permet de reporter toutes les modifications d'une autre branche, puis de réaliser d'autres modifications avant de réaliser une nouvelle validation.

À présent, vous pouvez envoyer au mainteneur un message indiquant que vous avez réalisé les modifications demandées et qu'il peut trouver cette nouvelle mouture sur votre branche `fonctionBv2` (voir figure 5-18).

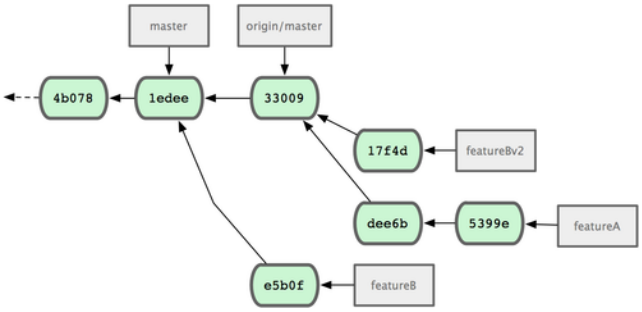

Figure 5.18: *Historique des validations après le travail sur fonctionBv2.*

5.2.5 Grand projet public

De nombreux grands projets ont des procédures établies pour accepter des patchs — il faut vérifier les règles spécifiques à chaque projet qui peuvent varier. Néanmoins, ils sont nombreux à accepter les patchs via une liste de diffusion de développement, ce que nous allons éclairer d'un exemple.

La méthode est similaire au cas précédent — vous créez une branche thématique par série de patchs sur laquelle vous travaillez. La différence réside dans la manière de les soumettre au projet. Au lieu de dupliquer le projet et de pousser vos soumissions sur votre dépôt, il faut générer des versions e-mail de chaque série de commits et les envoyer à la liste de diffusion de développement.

```
$ git checkout -b sujetA
$ (travail)
$ git commit
```

```
$ (travail)
$ git commit
```

Vous avez à présent deux commits que vous souhaitez envoyer à la liste de diffusion. Vous utilisez `git format-patch` pour générer des fichiers au format mbox que vous pourrez envoyer à la liste. Cette commande transforme chaque commit en un message e-mail dont le sujet est la première ligne du message de validation et le corps contient le reste du message plus le patch correspondant. Un point intéressant de cette commande est qu'appliquer le patch à partir d'un e-mail formaté avec `format-patch` préserve toute l'information de validation comme nous le verrons dans le chapitre suivant lorsqu'il s'agit de l'appliquer.

```
$ git format-patch -M origin/master
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
```

La commande `format-patch` affiche les noms de fichiers de patch créés. L'option `-M` indique à Git de suivre les renommages. Le contenu des fichiers ressemble à ceci :

```
$ cat 0001-Ajout-d-une-limite-la-fonction-de-log.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Ajout d'une limite à la fonction de log

Limite la fonctionnalité de log aux 20 premières lignes

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
1.6.2.rc1.20.g8c5b.dirty
```

Vous pouvez maintenant éditer ces fichiers de patch pour ajouter plus d'information à destination de la liste de diffusion mais que vous ne souhaitez par voir apparaître dans le message de valida-

tion. Si vous ajoutez du texte entre la ligne `--` et le début du patch (la ligne `lib/simplegit.rb`), les développeurs peuvent le lire mais l'application du patch ne le prend pas en compte.

Pour envoyer par e-mail ces fichiers, vous pouvez soit copier leur contenu dans votre application d'e-mail ou l'envoyer via une ligne de commande. Le copier-coller cause souvent des problèmes de formatage, spécialement avec les applications « intelligentes » qui ne préservent pas les retours à la ligne et les types d'espace. Heureusement, Git fournit un outil pour envoyer correctement les patchs formatés via IMAP, la méthode la plus facile. Je démontrerai comment envoyer un patch via Gmail qui s'avère être l'agent e-mail que j'utilise ; vous pourrez trouver des instructions détaillées pour de nombreuses applications de mail à la fin du fichier susmentionné `Documentation/SubmittingPatches` du code source de Git.

Premièrement, il est nécessaire de paramétrer la section imap de votre fichier `~/.gitconfig`. Vous pouvez positionner ces valeurs séparément avec une série de commandes `git config`, ou vous pouvez les ajouter manuellement. À la fin, le fichier de configuration doit ressembler à ceci :

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Si votre serveur IMAP n'utilise pas SSL, les deux dernières lignes ne sont probablement pas nécessaires et le paramètre `host` commencera par `imap://` au lieu de `imaps://`. Quand c'est fait, vous pouvez utiliser la commande `git send-email` pour placer la série de patchs dans le répertoire *Drafts* du serveur IMAP spécifié :

```
$ git send-email *.patch
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

La première question demande l'adresse mail d'origine (avec par défaut celle saisie en config), tandis que la seconde demande les destinataires. Enfin la dernière question sert à indiquer que l'on souhaite poster la série de patchs comme une réponse au premier patch de la série, créant ainsi un fil de discussion unique pour cette série. Ensuite, Git crache un certain nombre d'informations qui ressemblent à ceci pour chaque patch à envoyer :

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
```

```
To: jessica@example.com
Subject: [PATCH 1/2] Ajout d'une limite à la-fonction de log
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

À présent, vous devriez pouvoir vous rendre dans le répertoire Drafts, changer le champ destinataire pour celui de la liste de diffusion, y ajouter optionnellement en copie le mainteneur du projet ou le responsable et l'envoyer.

5.2.6 Résumé

Ce chapitre a traité quelques unes des méthodes communes de gestion de types différents de projets Git que vous pourrez rencontrer et a introduit un certain nombre de nouveaux outils pour vous aider à gérer ces processus. Dans la section suivante, nous allons voir comment travailler de l'autre côté de la barrière : en tant que mainteneur de projet Git. Vous apprendrez comment travailler comme dictateur bénévole ou gestionnaire d'intégration.

5.3 Maintenance d'un projet

En plus de savoir comment contribuer efficacement à un projet, vous aurez probablement besoin de savoir comment en maintenir un. Cela peut consister à accepter et appliquer les patchs générés via `format-patch` et envoyés par e-mail, ou à intégrer des modifications dans des branches distantes de dépôts distants. Que vous mainteniez le dépôt de référence ou que vous souhaitiez aider en vérifiant et approuvant les patchs, vous devez savoir comment accepter les contributions d'une manière limpide pour vos contributeurs et soutenable à long terme pour vous.

5.3.1 Travail dans des branches thématiques

Quand vous vous apprêtez à intégrer des contributions, un bonne idée consiste à les essayer d'abord dans une branche thématique, une branche temporaire spécifiquement créée pour essayer cette nouveauté. De cette manière, il est plus facile de rectifier un patch à part et de le laisser s'il ne fonctionne pas jusqu'à ce que vous disposiez de temps pour y travailler. Si vous créez une simple branche nommée d'après le thème de la modification que vous allez essayer, telle que `ruby_client` ou quelque chose d'aussi descriptif, vous pouvez vous en souvenir simplement plus tard. Le mainteneur du projet Git a l'habitude d'utiliser des espaces de nommage pour ses branches, tels que `sc/ruby_client`, où `sc` représente les initiales de la personne qui a contribué les modifications. Comme vous devez vous en souvenir, on crée une branche à part du master de la manière suivante :

```
$ git branch sc/ruby_client master
```

Ou bien, si vous voulez aussi basculer immédiatement dessus, vous pouvez utiliser l'option `checkout -b` :

```
$ git checkout -b sc/ruby_client master
```

Vous voilà maintenant prêt à ajouter les modifications sur cette branche thématique et à déterminer si c'est prêt à être fusionné dans les branches au long cours.

5.3.2 Application des patchs à partir d'e-mail

Si vous recevez par e-mail un patch que vous devez intégrer à votre projet, vous avez besoin d'appliquer le patch dans une branche thématique pour l'évaluer. Il existe deux méthodes pour appliquer un patch envoyé par e-mail : `git apply` et `git am`.

Application d'un patch avec apply

Si vous avez reçu le patch de quelqu'un qui l'a généré avec la commande `git diff` ou `diff` Unix, vous pouvez l'appliquer avec la commande `git apply`. Si le patch a été sauvé comme fichier `/tmp/patch-ruby-client.patch`, vous pouvez l'appliquer comme ceci :

```
$ git apply /tmp/patch-ruby-client.patch
```

Les fichiers dans votre copie de travail sont modifiés. C'est quasiment identique à la commande `patch -p1` qui applique directement les patchs mais en plus paranoïaque et moins tolérant sur les concordances approximatives. Les ajouts, effacements et renommages de fichiers sont aussi gérés s'ils sont décrits dans le format `git diff`, ce que `patch` ne supporte pas. Enfin, `git apply` fonctionne en mode « applique tout ou refuse tout » dans lequel toutes les modifications proposées sont appliquées si elles le peuvent, sinon rien n'est modifié là où `patch` peut n'appliquer que partiellement les patchs, laissant le répertoire de travail dans un état intermédiaire. `git apply` est par dessus tout plus paranoïaque que `patch`. Il ne créera pas une validation à votre place : après l'avoir lancé, vous devrez indexer et valider les modifications manuellement.

Vous pouvez aussi utiliser `git apply` pour voir si un patch s'applique proprement avant de réellement l'appliquer — vous pouvez lancer `git apply --check` avec le patch :

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

S'il n'y pas de message, le patch devrait s'appliquer proprement. Cette commande se termine avec un statut non-nul si la vérification échoue et vous pouvez donc l'utiliser dans des scripts.

Application d'un patch avec am

Si le contributeur est un utilisateur de Git qui a été assez gentil d'utiliser la commande `format-patch` pour générer ses patchs, votre travail sera facilité car le patch contient alors déjà l'information

d'auteur et le message de validation. Si possible, encouragez vos contributeurs à utiliser `format-patch` au lieu de `patch` pour générer les patchs qu'ils vous adressent. Vous ne devriez avoir à n'utiliser `git apply` que pour les vrais patchs.

Pour appliquer un patch généré par `format-patch`, vous utilisez `git am`. Techniquement, `git am` s'attend à lire un fichier au format mbox, qui est un format texte simple permettant de stocker un ou plusieurs messages e-mail dans un unique fichier texte. Un fichier ressemble à ceci :

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Ajout d'une limite à la fonction de log

Limite la fonctionnalité de log aux 20 premières lignes
```

C'est le début de ce que la commande `format-patch` affiche, comme vous avez vu dans la section précédente. C'est aussi un format e-mail mbox parfaitement valide. Si quelqu'un vous a envoyé par e-mail un patch correctement formaté en utilisant `git send-mail` et que vous le téléchargez en format mbox, vous pouvez pointer `git am` sur ce fichier mbox et il commencera à appliquer tous les patchs contenus. Si vous utilisez un client e-mail qui sait sauver plusieurs messages au format mbox, vous pouvez sauver la totalité de la série de patchs dans un fichier et utiliser `git am` pour les appliquer tous en une fois.

Néanmoins, si quelqu'un a déposé un fichier de patch généré via `format-patch` sur un système de suivi de faits techniques ou quelque chose similaire, vous pouvez toujours sauvegarder le fichier localement et le passer à `git am` pour l'appliquer :

```
$ git am 0001-limite-la-fonction-de-log.patch
Applying:  Ajout d'une limite à la fonction de log
```

Vous remarquez qu'il s'est appliqué proprement et a créé une nouvelle validation pour vous. L'information d'auteur est extraite des entêtes `From` et `Date` tandis que le message de validation est repris du champ `Subject` et du corps (avant le patch) du message. Par exemple, si le patch est appliqué depuis le fichier mbox ci-dessus, la validation générée ressemblerait à ceci :

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

Ajout d'une limite à la fonction de log

Limite la fonctionnalité de log aux 20 premières lignes
```

L'information `Commit` indique la personne qui a appliqué le patch et la date d'application. L'information `Author` indique la personne qui a créé le patch et la date de création.

Il reste la possibilité que le patch ne s'applique pas proprement. Peut-être votre branche principale a déjà trop divergé de la branche sur laquelle le patch a été construit, ou le patch dépend d'un autre patch qui n'a pas encore été appliqué. Dans ce cas, le processus de `git am` échouera et vous demandera ce que vous souhaitez faire :

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Cette commande introduit des marqueurs de conflit dans tous les fichiers qui ont généré un problème, de la même manière qu'un conflit de fusion ou de rebasage. Vous pouvez résoudre les problèmes de manière identique — éditez le fichier pour résoudre les conflits, indexez le nouveau fichier, puis lancez `git am --resolved` pour continuer avec le patch suivant :

```
$ (correction du fichier)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Si vous souhaitez que Git essaie de résoudre les conflits avec plus d'intelligence, vous pouvez passer l'option `-3` qui demande à Git de tenter une fusion à trois sources. Cette option n'est pas active par défaut parce qu'elle ne fonctionne pas si le commit sur lequel le patch indique être basé n'existe pas dans votre dépôt. Si par contre, le patch est basé sur un commit public, l'option `-3` est généralement beaucoup plus fine pour appliquer des patchs conflictuels :

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

Dans ce cas, je cherchais à appliquer un patch qui avait déjà été intégré. Sans l'option `-3`, cela aurait ressemblé à un conflit.

Si vous appliquez des patchs à partir d'un fichier mbox, vous pouvez aussi lancer la commande `am` en mode interactif qui s'arrête à chaque patch trouvé et vous demande si vous souhaitez l'appliquer :

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

C'est agréable si vous avez un certain nombre de patchs sauvegardés parce que vous pouvez voir les patchs pour vous rafraîchir la mémoire et ne pas les appliquer s'ils ont déjà été intégrés.

Quand tous les patchs pour votre sujet ont été appliqués et validés dans votre branche, vous pouvez choisir si et comment vous souhaitez les intégrer dans une branche au long cours.

5.3.3 Vérification des branches distantes

Si votre contribution a été fournie par un utilisateur de Git qui a mis en place son propre dépôt public sur lequel il a poussé ses modifications et vous a envoyé l'URL du dépôt et le nom de la branche distante, vous pouvez les ajouter en tant que dépôt distant et réaliser les fusions localement.

Par exemple, si Jessica vous envoie un e-mail indiquant qu'elle a une nouvelle fonctionnalité géniale dans la branche `ruby-client` de son dépôt, vous pouvez la tester en ajoutant le dépôt distant et en tirant la branche localement :

```
$ git remote add jessica git://github.com/jessica/monprojet.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Si elle vous envoie un autre mail indiquant une autre branche contenant une autre fonctionnalité géniale, vous pouvez la récupérer et la tester simplement à partir de votre référence distante.

C'est d'autant plus utile si vous travaillez en continu avec une personne. Si quelqu'un n'a qu'un seul patch à contribuer de temps en temps, l'accepter via e-mail peut s'avérer moins consommateur en temps de préparation du serveur public, d'ajout et retrait de branches distantes juste pour tirer quelques patchs. Vous ne souhaiteriez sûrement pas devoir gérer des centaines de dépôts distants pour intégrer à chaque fois un ou deux patchs. Néanmoins, des scripts et des services hébergés peuvent rendre cette tâche moins ardue. Cela dépend largement de votre manière de développer et de celle de vos contributeurs.

Cette approche a aussi l'avantage de vous fournir l'historique des validations. Même si vous pouvez rencontrer des problèmes de fusion légitimes, vous avez l'information dans votre historique de la base ayant servi pour les modifications contribuées. La fusion à trois sources est choisie par défaut plutôt que d'avoir à spécifier l'option `-3` en espérant que le patch a été généré à partir d'un instantané public auquel vous auriez accès.

Si vous ne travaillez pas en continu avec une personne mais souhaitez tout de même tirer les modifications de cette manière, vous pouvez fournir l'URL du dépôt distant à la commande `git pull`. Cela permet de réaliser un tirage unique sans sauver l'URL comme référence distante :

```
$ git pull git://github.com/typeunique/projet.git
From git://github.com/typeunique/projet
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

5.3.4 Déterminer les modifications introduites

Vous avez maintenant une branche thématique qui contient les contributions. De ce point, vous pouvez déterminer ce que vous souhaitez en faire. Cette section revisite quelques commandes qui vont vous permettre de faire une revue de ce que vous allez exactement introduire si vous fusionnez dans la branche principale.

Faire une revue de tous les commits dans cette branche s'avère souvent d'une grande aide. Vous pouvez exclure les commits de la branche master en ajoutant l'option `--not` devant le nom de la branche. Par exemple, si votre contributeur vous envoie deux patchs et que vous créez une branche appelée `contrib` et y appliquez ces patchs, vous pouvez lancer ceci :

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Pour visualiser les modifications que chaque commit introduit, souvenez-vous que vous pouvez passer l'option `-p` à `git log` et elle ajoutera le diff introduit à chaque commit.

Pour visualiser un diff complet de ce qui arriverait si vous fusionniez cette branche thématique avec une autre branche, vous pouvez utiliser un truc bizarre pour obtenir les résultats corrects. Vous pourriez penser à lancer ceci :

```
$ git diff master
```

Cette commande affiche un diff mais elle peut être trompeuse. Si votre branche `master` a avancé depuis que vous en avez créé la branche thématique, vous obtiendrez des résultats apparemment étranges. Cela arrive parce que Git compare directement l'instantané de la dernière validation sur la branche thématique et celui de la dernière validation sur la branche `master`. Par exemple, si vous avez ajouté une ligne dans un fichier sur la branche `master`, une comparaison directe donnera l'impression que la branche thématique va retirer cette ligne.

Si `master` est un ancêtre directe de la branche thématique, ce n'est pas un problème. Si les deux historiques ont divergé, le diff donnera l'impression que vous ajoutez toutes les nouveautés de

la branche thématique et retirez tout ce qui a été fait depuis dans la branche `master`.

Ce que vous souhaitez voir en fait, ce sont les modifications ajoutées sur la branche thématique — le travail que vous introduirez si vous fusionnez cette branche dans master. Vous obtenez ce résultat en demandant à Git de comparer le dernier instantané de la branche thématique avec son ancêtre commun à la branch master le plus récent.

Techniquement, c'est réalisable en déterminant exactement l'ancêtre commun et en lançant la commande diff dessus :

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Néanmoins, comme ce n'est pas très commode, Git fournit un raccourci pour réaliser la même chose : la syntaxe à trois points. Dans le contexte de la commande `diff`, vous pouvez placer trois points après une autre branche pour réaliser un `diff` entre le dernier instantané de la branche sur laquelle vous vous trouvez et son ancêtre commun avec une autre branche :

```
$ git diff master...contrib
```

Cette commande ne vous montre que les modifications que votre branche thématique a introduites depuis son ancêtre commun avec master. C'est une syntaxe très utile à se souvenir.

5.3.5 Intégration des contributions

Lorsque tout le travail de votre branche thématique est prêt à être intégré dans la branche principale, il reste à savoir comment le faire. De plus, il faut connaître le mode de gestion que vous souhaitez pour votre projet. Vous avez de nombreux choix et je vais en traiter quelques uns.

Modes de fusion

Un mode simple fusionne votre travail dans la branche `master`. Dans ce scénario, vous avez une branche `master` qui contient le code stable. Quand vous avez des modifications prêtes dans une branche thématique, vous la fusionnez dans votre branche master puis effacez la branche thématique, et ainsi de suite. Si vous avez un dépôt contenant deux branches nommées `ruby_client` et `php_client` qui ressemble à la figure 5-19 et que vous fusionnez `ruby_client` en premier, suivi de `php_client`, alors votre historique ressemblera à la fin à la figure 5-20.

C'est probablement le mode le plus simple mais cela peut s'avérer problématique si vous avez à gérer des dépôts ou des projets plus gros.

Si vous avez plus de développeurs ou un projet plus important, vous souhaiterez probablement utiliser un cycle à fusion à au moins deux étapes. Dans ce scénario, vous avez deux branches au long cours, `master` et `develop`, dans lequel vous déterminez que `master` est mis à jour seulement lors d'une version vraiment stable et tout le nouveau code est intégré dans la branche `develop`. Vous poussez régulièrement ces deux branches sur le dépôt public. Chaque fois que vous avez une nouvelle branche thématique à fusionner (figure 5-21), vous la fusionnez dans `develop` (figure 5-22). Puis, lorsque vous balisez une version majeure, vous mettez `master` à niveau avec l'état stable de `develop` en avance rapide (figure 5-23).

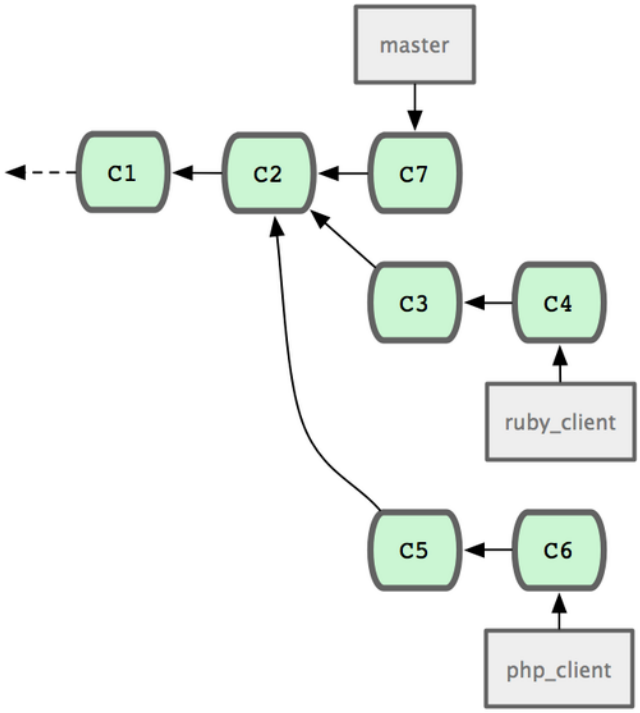

Figure 5.19: *Historique avec quelques branches thématiques.*

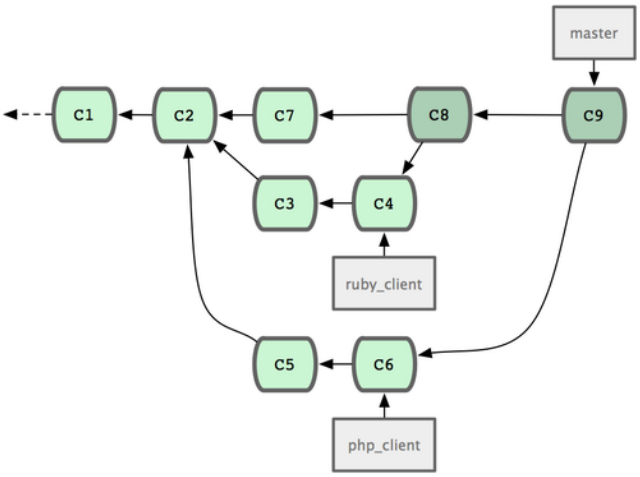

Figure 5.20: *Après fusion d'une branche thématique.*

Ainsi, lorsque l'on clone le dépôt de votre projet, on peut soit extraire la branche master pour construire la dernière version stable et mettre à jour facilement ou on peut extraire le branche develop qui représente le nec plus ultra du développement.

Vous pouvez aussi continuer ce concept avec une branche d'intégration où tout le travail est fusionné. Alors, quand la base de code sur cette branche est stable et que les tests passent, vous la fusionnez dans la branche develop. Quand cela s'est avéré stable pendant un certain temps, vous mettez à jour la branche master en avance rapide.

Gestions avec nombreuses fusions

Le projet Git dispose de quatre branches au long cours : `master`, `next`, `pu` (proposed updates : propositions) pour les nouveaux travaux et `maint` pour les backports de maintenance. Quand une nouvelle contribution est proposée, elle est collectée dans des branches thématiques dans le dépôt

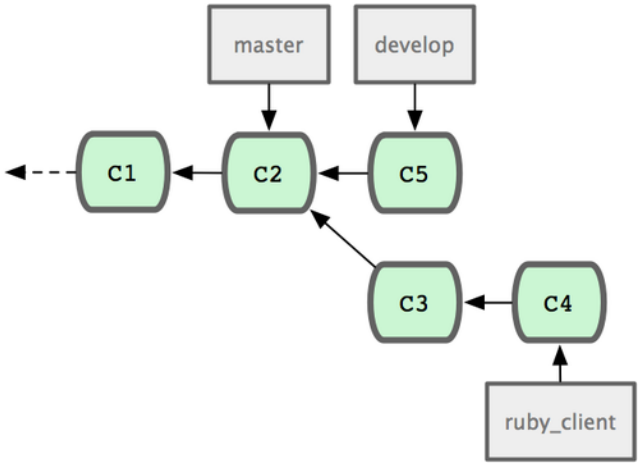

Figure 5.21: *Avant la fusion d'une branche thématique.*

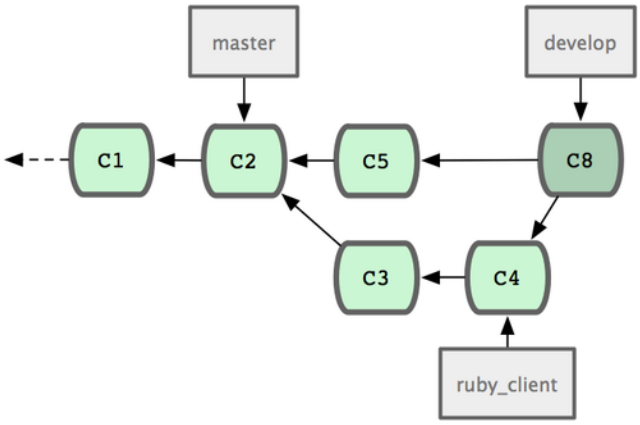

Figure 5.22: *Après la fusion d'une branche thématique.*

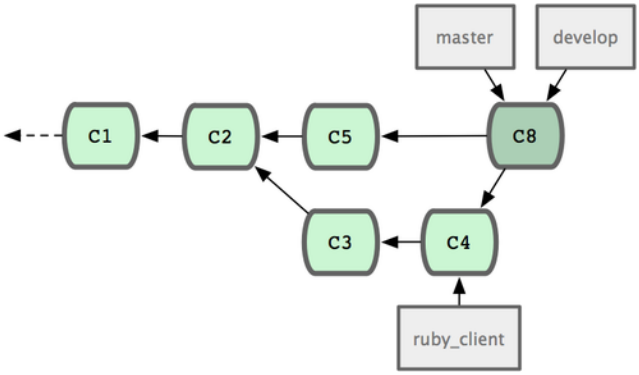

Figure 5.23: *Après une publication d'une branche thématique.*

du mainteneur d'une manière similaire à ce que j'ai décrit (voir figure 5-24). A ce point, les fonctionnalités sont évaluées pour déterminer si elles sont stables et prêtes à être consommées ou si elles nécessitent un peaufinage. Si elles sont stables, elles sont fusionnées dans `next` et cette branche est poussée sur le serveur public pour que tout le monde puisse essayer les fonctionnalités intégrées ensemble.

Si les fonctionnalités nécessitent encore du travail, elles sont fusionnées plutôt dans `pu`. Quand elles sont considérées comme totalement stables, elles sont re-fusionnées dans `master` et sont alors reconstruites à partir fonctionnalités qui résidaient dans `next` mais n'ont pu intégrer `master`. Cela

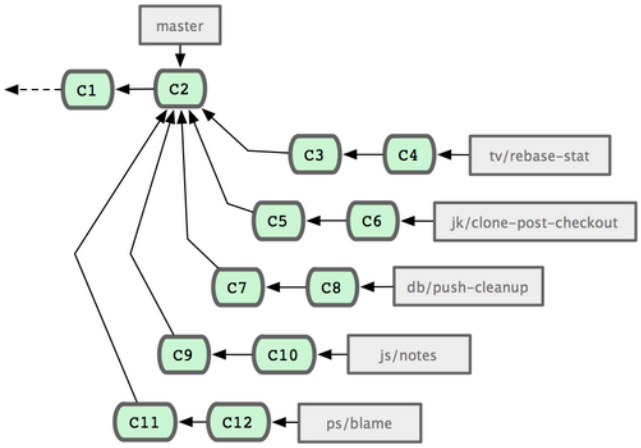

Figure 5.24: *Série complexe de branches thématiques contribuées en parallèle.*

signifie que `master` évolue quasiment toujours en mode avance rapide, tandis que `next` est rebasé assez souvent et `pu` est rebasé encore plus souvent (voir figure 5-25).

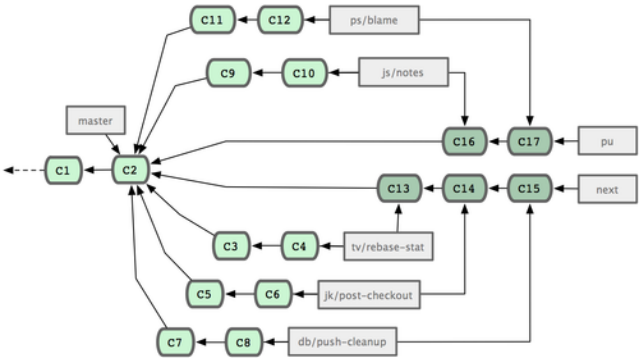

Figure 5.25: *Fusion des branches thématiques dans les branches à long terme.*

Quand une branche thématique a finalement été fusionnée dans `master`, elle est effacée du dépôt. Le projet Git a aussi une branche `maint` qui est créée à partir de la dernière version pour fournir des patchs correctifs en cas de besoin de version de maintenance. Ainsi, quand vous clonez le dépôt de Git, vous avez quatre branches disponibles pour évaluer le projet à différentes étapes de développement, selon le niveau développement que vous souhaitez utiliser ou pour lequel vous souhaitez contribuer. Le mainteneur a une gestion structurée qui lui permet d'évaluer et sélectionner les nouvelles contributions.

Gestion par rebasage et sélection de commit

D'autres mainteneurs préfèrent rebaser ou sélectionner les contributions sur le sommet de la branche master, plutôt de les fusionner, de manière à conserver un historique à peu près linéaire. Lorsque plusieurs modifications sont présentes dans une branche thématique et que vous souhaitez les intégrer, vous vous placez sur cette branche et vous lancer la commande rebase pour reconstruire les modifications à partir du sommet courant de la branche `master` (ou `develop`, ou autre). Si cela fonctionne correctement, vous pouvez faire une avance rapide sur votre branche `master` et vous obtenez au final un historique de projet linéaire.

L'autre moyen de déplacer des modifications introduites dans une branche vers une autre consiste à les sélectionner (`cherry-pick`). Une sélection dans Git ressemble à un rebasage appliqué

à un commit unique. Cela consiste à prendre le patch qui a été introduit lors d'une validation et à essayer de l'appliquer sur la branche sur laquelle on se trouve. C'est très utile si on a un certain nombre de commits sur une branche thématique et que l'on veut n'en intégrer qu'un seul, ou si on n'a qu'un commit sur une branche thématique et qu'on préfère le sélectionner plutôt que de lancer rebase. Par exemple, supposons que vous ayez un projet ressemblant à la figure 5-26.

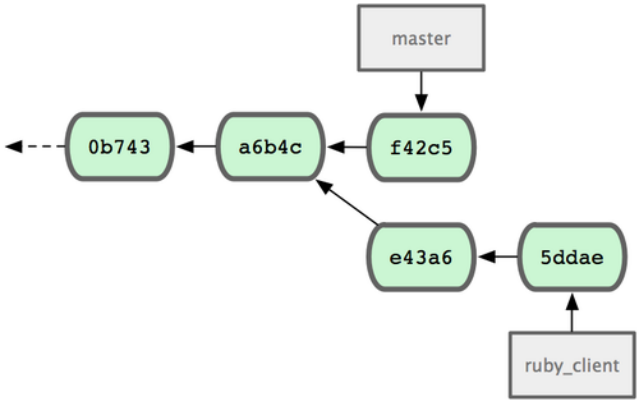

Figure 5.26: *Historique d'exemple avant une sélection.*

Si vous souhaitez tirer le commit `e43a6` dans votre branche master, vous pouvez lancer

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

La même modification que celle introduite en `e43a6` est tirée mais vous obtenez une nouvelle valeur de SHA-1 car les dates d'application sont différentes. À présent, votre historique ressemble à la figure 5-27.

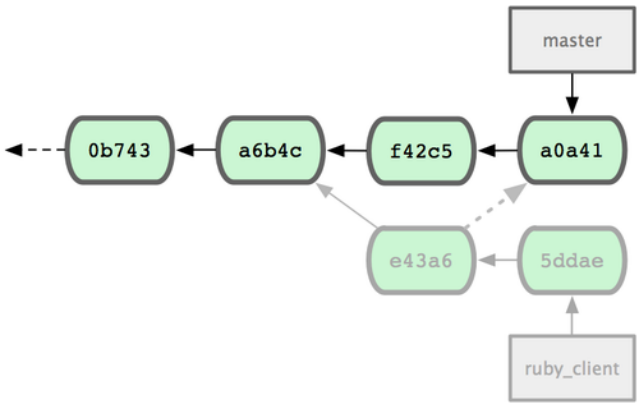

Figure 5.27: *Historique après sélection d'un commit dans une branche thématique.*

Maintenant, vous pouvez effacer votre branche thématique et abandonner les commits que vous n'avez pas tirés dans master.

5.3.6 Balisage de vos publications

Quand vous décidez d'arrêter une publication de votre projet, vous souhaiterez probablement baliser le projet pour pouvoir recréer cette version dans le futur. Vous pouvez créer une nouvelle balise telle que décrite au chapitre 2. Si vous décidez de signer la balise en tant que mainteneur, la commande ressemblera à ceci :

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Si vous signez vos balises, vous rencontrerez le problème de la distribution de votre clé publique PGP permettant de vérifier la signature. Le mainteneur du projet Git a résolu le problème en incluant la clé publique comme blob dans le dépôt et en ajoutant une balise qui pointe directement sur ce contenu. Pour faire de même, vous déterminez la clé de votre trousseau que vous voulez publier en lançant `gpg --list-keys` :

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Ensuite, vous pouvez importer la clé directement dans la base de donnée Git en l'exportant de votre trousseau et en la redirigeant dans `git hash-object` qui écrit une nouveau blob avec son contenu dans Git et vous donne en sortie le SHA-1 du blob :

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

À présent, vous avez le contenu de votre clé dans Git et vous pouvez créer une balise qui pointe directement dessus en spécifiant la valeur SHA-1 que la commande `hash-object` vous a fournie :

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si vous lancez `git push --tags`, la balise `mainteneur-pgp-pub` sera partagée publiquement. Un tiers pourra vérifier une balise après import direct de votre clé publique PGP, en extrayant le blob de la base de donnée et en l'important dans GPG :

```
$ git show maintainer-pgp-pub | gpg --import
```

Il pourra alors utiliser cette clé pour vérifier vos balises signées. Si de plus, vous incluez des instructions d'utilisation pour la vérification de signature dans le message de balisage, l'utilisateur aura accès à ces informations en lançant la commande `git show <balise>`.

5.3.7 Génération d'un nom de révision

Comme Git ne fournit pas par nature de nombres croissants tels que « r123 » à chaque validation, la commande `git describe` permet de générer un nom humainement lisible pour chaque commit. Git concatène la nom de la balise la plus proche, le nombre de validations depuis cette balise et un code SHA-1 partiel du commit que l'on cherche à définir :

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

De cette manière, vous pouvez exporter un instantané ou le construire et le nommer de manière intelligible. En fait, si Git est construit à partir du source cloné depuis le dépôt Git, `git --version` vous donne exactement cette valeur. Si vous demandez la description d'un instantané qui a été balisé, le nom de la balise est retourné.

La commande `git describe` repose sur les balises annotées (balises créées avec les options `-a` ou `-s`). Les balises de publication doivent donc être créées de cette manière si vous souhaitez utiliser `git describe` pour garantir que les commits seront décrits correctement. vous pouvez aussi utiliser ces noms comme cible lors d'une extraction ou d'une commande `show`, bien qu'ils reposent sur le SHA-1 abrégé et pourraient ne pas rester valide indéfiniment. Par exemple, le noyau Linux a sauté dernièrement de 8 à 10 caractères pour assurer l'unicité des objets SHA-1 et les anciens noms `git describe` sont par conséquent devenus invalides.

5.3.8 Préparation d'une publication

Maintenant, vous voulez publier une version. Une des étapes consiste à créer une archive du dernier instantané de votre code pour les pauvres hères qui n'utilisent pas Git. La commande dédiée à cette action est `git archive` :

```
$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Lorsqu'on ouvre l'archive, on obtient le dernier instantané du projet sous un répertoire `projet`. On peut aussi créer une archive au format zip de manière similaire en passant l'option `--format=zip` à la commande `git archive` :

```
$ git archive master --prefix='projet/' --format=zip > `git describe master`.zip
```

Voilà deux belles archives tar.gz et zip de votre projet prêtes à être téléchargées sur un site web ou envoyées par e-mail.

5.3.9 Shortlog

Il est temps d'envoyer une annonce à la liste de diffusion des annonces relatives à votre projet. Une manière simple d'obtenir rapidement une sorte de liste des modifications depuis votre dernière version ou e-mail est d'utiliser la commande `git shortlog`. Elle résume toutes le validations dans l'intervalle qui vous lui spécifiez. Par exemple, ce qui suit vous donne un résumé de toutes les validations depuis votre dernière version si celle-ci se nomme v1.0.1 :

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

Vous obtenez ainsi un résumé clair de toutes les validations depuis v1.0.1, regroupées par auteur, prêt à être envoyé sur la liste de diffusion.

5.4 Résumé

Vous devriez à présent vous sentir à l'aise pour contribuer à un projet avec Git, mais aussi pour maintenir votre propre projet et intégrer les contributions externes. Félicitations, vous êtes un développeur Git efficace ! Au prochain chapitre, vous découvrirez des outils plus puissants pour gérer des situations complexes, qui feront de vous un maître de Git.

Chapitre 6

Utilitaires Git

A présent, vous avez appris les commandes et modes de fonctionnements usuels requis pour gérer et maintenir un dépôt Git pour la gestion de votre code source. Vous avez déroulé les routines de suivi (*tracking*) et de consignation (*committing*) de fichiers, vous avez exploité la puissance de la zone d'attente (*staging area*), de la création et de la fusion de branches locales de travail.

Maintenant, vous allez explorer un certain nombre de fonctionnalités particulièrement efficaces, fonctionnalités que vous n'utiliserez que rarement mais dont vous pourriez avoir l'usage à un moment ou à un autre.

6.1 Sélection des versions

Git vous permet d'adresser certains *commits* ou un ensemble de *commits* de différentes façons. Si elles ne sont pas toutes évidentes il est bon de les connaître.

6.1.1 Révisions ponctuelles

Naturellement, vous pouvez référencer un commit par la signature SHA-1, mais il existe des méthodes plus confortables pour le genre humain. Cette section présente les méthodes pour référencer un commit simple.

6.1.2 Empreinte SHA courte

Git est capable de deviner de quel commit vous parlez si vous ne fournissez que quelques caractères au début de la signature, tant que votre SHA-1 partiel comporte au moins 4 caractères et ne génère pas de collision. Dans ces conditions, un seul objet correspondra à ce SHA-1.

Par exemple, pour afficher un commit précis, supposons que vous exécutiez `git log` et que vous identifiez le commit où vous avez introduit une fonctionnalité précise.

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests
```

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

Pour cet exemple, choisissons `1c002dd....` Si vous affichez le contenu de ce commit via `git show`, les commandes suivantes sont équivalentes (en partant du principe que les SHA-1 courts ne sont pas ambigüs).

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git peut déterminer un SHA tout à la fois le plus court possible et non ambigü. Ajoutez l'option `--abbrev-commit` à la commande `git log` et le résultat affiché utilisera des valeurs plus courtes mais uniques; par défaut git retiendra 7 caractères et augmentera au besoin :

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

En règle générale, entre 8 et 10 caractères sont largement suffisant pour assurer l'unicité dans un projet. Un des plus gros projets utilisant Git, le kernel Linux, nécessite de plus en plus fréquemment 12 caractères sur les 40 possibles pour assurer l'unicité.

6.1.3 QUELQUES MOTS SUR SHA-1

Beaucoup de gens se soucient qu'à un moment donné ils auront, par des circonstances hasardeuses, deux objets dans leur référentiel de hachage de même empreinte SHA-1. Qu'en est-il réellement ?

S'il vous arrivait de consigner (*commit*) un objet qui se hache de la même empreinte SHA-1 d'un objet existant dans votre référentiel, Git verrez l'objet existant déjà dans votre base de données et Git présumera qu'il était déjà enregistré. Si vous essayez de récupérer l'objet de nouveau à un moment donné, vous aurez toujours les données du premier objet.

Quoi qu'il en soit, vous devriez être conscient à quel point ce scénario est ridiculement improbable. Une empreinte SHA-1 porte sur 20 octets soit 160bits. Le nombre d'objet aléatoires à hasher requis pour assurer une probabilité de collision de 50% vaut environ 2^{80} (la formule pour calculer

la probabilité de collision est $p = \frac{n(n-1)}{2} \times \frac{1}{2^{160}}$). 2^{80} vaut 1.2×10^{24} soit 1 million de milliards de milliards. Cela représente 1200 fois le nombre de grains de sable sur terre.

Voici un exemple pour vous donner une idée de ce qui pourrait provoquer une collision du SHA-1. Si tous les 6,5 milliards d'humains sur Terre programmait et que chaque seconde, chacun produisait du code équivalent à l'historique entier du noyaux Linux (1 million d'objets Git) et le poussait sur un énorme dépôt Git, cela prendrait 5 ans pour que ce dépôt contienne assez d'objets pour avoir une probabilité de 50% qu'une seule collision SHA-1 existe. Il y a une probabilité plus grande que tous les membres de votre équipe de programmation serait attaqués et tués par des loups dans des incidents sans relation la même nuit.

6.1.4 Références de branches

La méthode la plus commune pour désigner un commit est une branche y pointant. Dès lors, vous pouvez utiliser le nom de la branche dans toute commande utilisant un objet de type commit ou un SHA-1. Par exemple, si vous souhaitez afficher le dernier commit d'une branche, les commandes suivantes sont équivalentes, en supposant que la branche `sujet1` pointe sur `ca82a6d` :

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show sujet1
```

Pour connaître l'empreinte SHA sur lequel pointe une branche, ou pour savoir parmi tous les exemples précédents ce que cela donne en terme de SHA, vous pouvez utiliser la commande de plomberie nommée `rev-parse`. Se référer au chapitre 9 pour plus d'informations sur les commandes de plomberie; en résumé, `rev-parse` est là pour les opérations de bas niveau et n'est pas conçue pour être utilisée au jour le jour. Quoi qu'il en soit, cela peut se révéler utile pour comprendre ce qui se passe. Je vous invite à tester `rev-parse` sur votre propre branche.

```
$ git rev-parse sujet1
ca82a6dff817ec66f44342007202690a93763949
```

6.1.5 Raccourcis RefLog

Git maintient en arrière-plan un historique des références où sont passées HEAD et vos branches sur les dernieres mois - ceci s'appelle le reflog.

Vous pouvez le consulter avec la commande `git reflog` :

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

À chaque fois que l'extrémité de votre branche est modifiée, Git enregistre cette information pour vous dans son historique temporaire. Vous pouvez référencer d'anciens commits avec cette donnée. Si vous souhaitez consulter le n-ième antécédent de votre HEAD, vous pouvez utiliser la référence `@n` du reflog, 5 dans cet exemple :

```
$ git show HEAD@{5}
```

Vous pouvez également remonter le temps et savoir où en était une branche à un moment donné. Par exemple, pour savoir où en était la branche `master` hier (yesterday en anglais), tapez :

```
$ git show master@{yesterday}
```

Cette technique fonctionne uniquement si l'information est encore présente dans le reflog, vous ne pourrez donc pas consulter les commits trop anciens.

Pour consulter le reflog au format `git log`, exécutez: `git log -g` :

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Veuillez noter que le reflog ne stocke que des informations locales, c'est un historique de ce que vous avez fait dans votre dépôt. Les références ne sont pas copiées dans un autre dépôt; et juste après le clone d'un dépôt, votre reflog sera vide, puisque qu'aucune activité ne s'y sera produite. Exécuter `git show` HEAD@{2.months.ago}' ne fonctionnera que si vous avez dupliqué ce projet depuis au moins 2 mois — si vous l'avez dupliqué il y a 5 minutes, vous n'obtiendrez rien.

6.1.6 Références passées

Une solution fréquente de référencer un commit est d'utiliser son ancêtre. Si vous suffixez une référence par `^`, Git la résoudra comme étant le parent de cette référence. Supposons que vous consultiez votre historique :

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fix sur la gestion des refs, ajout gc auto, mise à jour des tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b modifs minor rdoc
* | 1c002dd ajout blame and merge
|/
* 1c36188 ignore *.gem
* 9b29157 ajout open3_detach à la liste des fichiers gemspcec
```

Alors, vous pouvez consulter le commit précédent en spécifiant `HEAD^`, ce qui signifie « le parent de HEAD » :

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Vous pouvez également spécifier un nombre après `^` — par exemple, `d921970^2` signifie « le second parent de d921970. ». Cette syntaxe ne sert que pour les commits de fusion, qui ont plus d'un parent. Le premier parent est la branche où vous avez fusionné, et le second est le commit de la branche que vous avez fusionnée :

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    ajout blame and merge

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    modifs minor rdoc
```

Une autre solution courante pour spécifier une référence est le `~`. Il fait également référence au premier parent, donc `HEAD~` et `HEAD^` sont équivalents. La différence se fait sentir si vous spécifiez un nombre. `HEAD~2` signifie « le premier parent du premier parent, » ou bien « le grand-parent »; ça remonte les premiers parents autant de fois que demandé. Par exemple, dans l'historique précédemment présenté, `HEAD~3` serait :

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Cela aura bien pu être écrit `HEAD^^^`, qui là encore est le premier parent du premier parent du premier parent :

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Vous pouvez également combiner ces syntaxes — vous pouvez obtenir le second parent de la référence précédente (en supposant que c'était un commit de fusion) en utilisant `HEAD~3^2`, etc.

6.1.7 Plages de commits

A présent que vous pouvez spécifier des commits individuels, voyons comme spécifier une place de commits. Ceci est particulièrement pratique pour la gestion des branches — si vous avez beaucoup de branches, vous pouvez utiliser les plages pour adresser des problèmes tels que « Quelle activité sur cette branche n'ai-je pas encore fusionné sur ma branche principale ? ».

Double point

La plus fréquente spécification de plage de commit est la syntaxe double-point. En gros, cela demande à Git de résoudre la plage des commits qui sont accessible depuis un commit mais ne le sont pas depuis un autre. Par exemple, disons que votre historique ressemble à celui de la Figure 6.1.

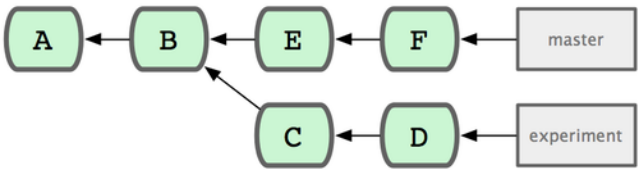

Figure 6.1: *Exemple d'historique pour la sélection de plages de commits.*

Si vous voulez savoir ce que n'a pas encore été fusionné sur votre branche master depuis votre branche `experiment`, vous pouvez demandez à Git de vous montrer un listing des commits avec `master..experiment` — ce qui signifie « tous les commits accessibles par experiment qui ne le sont pas par master. ». Dans un souci de brièveté et de clarté de ces exemples, je vais utiliser les lettres des commits issus du diagramme à la place du vrai listing dans l'ordre où ils auraient dû être affichés :

```
$ git log master..experiment
D
C
```

D'un autre côté, si vous souhaitez voir l'opposé — tous les commits dans `master` mais pas encore dans `experiment` — vous pouvez inverser les noms de branches, `experiment..master` vous montre tout ce que `master` accède mais qu'experiment ne voit pas :

```
$ git log experiment..master
F
E
```

C'est pratique si vous souhaitez maintenir `experiment` à jour et anticiper les fusions. Une autre cas d'utilisation fréquent et de voir ce que vous vous appréter à pousser sur une branche distante :

```
$ git log origin/master..HEAD
```

Cette commande vous affiche tous les commits de votre branche courante qui ne sont pas sur la branche `master` du dépôt distant `origin`. Si vous exécutez `git push` et que votre branche courante suit `origin/master`, les commits listés par `git log origin/master..HEAD` sont les commits qui seront transférés sur le serveur. Vous pouvez également laisser tomber une borne de la syntaxe pour faire comprendre à Git que vous parlez de HEAD. Par exemple, vous pouvez obtenir les mêmes résultats que précédemment en tapant `git log origin/master..` — Git utilise HEAD si une des bornes est manquante.

Emplacements multiples

La syntaxe double-point est pratique comme raccourci; mais peut-être souhaitez-vous utiliser plus d'une branche pour spécifier une révision, comme pour voir quels commits sont dans plusieurs branches mais qui sont absents de la branche courante. Git vous permets cela avec `^` or `--not` en préfixe de toute référence de laquelle vous ne souhaitez pas voir les commits. Les 3 commandes ci-après sont équivalentes :

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

C'est utile car cela vous permets de spécifier plus de 2 références dans votre requête, ce que vous ne pouvez accomplir avec la syntaxe double-point. Par exemple, si vous souhaitez voir les commits qui sont accessibles depuis `refA` et `refB` mais pas depuis `refC`, vous pouvez taper ces 2 commandes :

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Ceci vous fournit un système de requêtage des révisions très puissant, pour vous aider à saisir ce qui se trouve sur vos branches.

Triple point

La dernière syntaxe majeure de sélection de plage de commits est la syntaxe triple-point, qui spécifie tous les commits accessible par l'une des deux référence, exclusivement. Toujours avec l'exemple d'historique à la figure 6-1, si vous voulez voir ce qui ce trouve sur `master` ou `experiment` mais pas sur les 2, exécutez :

```
$ git log master...experiment
F
E
D
C
```

Encore une fois, cela vous donne un `log` normal mais ne vous montre les informations que pour ces quatre commits, dans l'ordre naturel des dates de commit.

Une option courante à utiliser avec la commande `log` dans ce cas est `--left-right` qui vous montre de quelle borne de la plage ce commit fait partie. Cela rend les données plus utiles :

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Avec ces outils, vous pourrez utiliser Git pour savoir quels commits inspecter.

6.2 Mise en attente interactive

Git propose quelques scripts qui rendent les opérations en ligne de commande plus simple. Nous allons à présent découvrir des commandes interactives vous permettant de choisir les fichiers ou une partie d'un fichier à incorporer à un commit. Ces outils sont particulièrement pratiques si vous modifiez un large périmètre de fichiers et que vous souhaitez les commiter séparement plutôt que massivement. De la sorte, vous vous assurez que vos commits sont des ensembles cohérents et qu'ils peuvent être facilement revus par vos collaborateurs. Si vous exécutez `git add` avec l'option `-i` ou `--interactive`, Git rentre en mode interactif, affichant quelque chose comme :

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

Vous vous apercevrez que cette commande propose une vue bien différente de votre zone d'attente; en gros, c'est la même information que vous auriez obtenue avec `git status` mais en plus succint et plus instructif. Cela liste les modifications que vous avez mises en attente à gauche, et celles en cours à droite.

En dessous vient la section des commandes (* **Commands***). Vous pourrez y faire bon nombre de choses, notamment mettre en attente des fichiers, les enlever de la zone d'attente, mettre en attente des parties de fichiers, ajouter des fichiers non indexés, et vérifier les différences de ce que vous avez mis en attente.

6.2.1 Mettre en attente des fichiers

Si vous tapez `2` ou `u` au prompt `What now>`, le script vous demande quels fichiers vous voulez mettre en attente :

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Pour mettre en attente les fichiers TODO et index.html, vous pouvez taper ces nombres :

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Le caractère `*` au début de la ligne de chaque fichier indique que celui-ci est sélectionné. Si vous tapez Entrée sur une invite `Update>>` vide, Git prend tout ce qui est sélectionné et le met en attente pour vous :

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

À présent, vous pouvez voir que les fichiers TODO et index.html sont mis en attente (staged en anglais) et que simplgit.rb ne l'est toujours pas. Si vous souhaitez enlever de la zone d'attente le fichier TODO, utilisez `3` (ou `r` pour revert en anglais) :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Un aperçu rapide à votre statut Git et vous pouvez voir que vous avez enlever de la zone d'attente le fichier TODO :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Pour voir la modification que vous avez mise en attente, utilisez `6` ou `d` (pour diff en anglais). Cela vous affiche la liste des fichiers en attente et vous pouvez choisir ceux pour lesquels vous voulez consulter la différence. C'est équivalent à `git diff --cached` en ligne de commande :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

Avec ces commandes élémentaires, vous pouvez utiliser l'ajout interactif pour manipuler votre zone d'attente un peu plus facilement.

6.2.2 Patches de Staging

Git est également capable de mettre en attente certaines parties d'un fichier. Par exemple, si vous modifiez en 2 endroits votre fichier simplegit.rb et que vous souhaitez mettre en attente l'une d'entre elles seulement, cela peut se faire très aisément avec Git. En mode interactif, tapez `5` ou `p` (pour patch en anglais). Git vous demandera quels fichiers vous voulez mettre en attente partiellement, puis, pour chaque section des fichiers sélectionnés, il affichera les parties de fichiers où il y a des différences et vous demandera si vous souhaitez les mettre en attente, un par un :

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

A cette étape, vous disposez de bon nombre d'options. `?` vous liste les actions possibles, voici une traduction :

```
Mettre en attente cette partie [y,n,a,d,/,j,J,g,e,?]? ?
y - mettre en attente cette partie
n - ne pas mettre en attente cette partie
a - mettre en attente cette partie et toutes celles restantes dans ce fichier
d - ne pas mettre en attente cette partie ni aucune de celles restantes dans ce fichier
g - sélectionner un partie à voir
/ - chercher une partie correspondant à la regexp donnée
j - laisser cette partie non décidée, voir la prochaine partie non encore décidée
J - laisser cette partie non décidée, voir la prochaine partie
k - laisser cette partie non décidée, voir la partie non encore décidée précendente
K - laisser cette partie non décidée, voir la partie précédente
s - couper la partie courante en parties plus petites
e - modifier manuellement la partie courante
? - afficher l'aide
```

En règle générale, vous choisirez `y` ou `n` pour mettre en attente ou non chacun des blocs, mais tout mettre en attente pour certains fichiers ou remettre à plus tard le choix pour un bloc peut également être utile. Si vous mettez en attente une partie d'un fichier et laissez une autre partie non en attente, vous statut ressemblera à peu près à ceci :

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

Le statut pour le fichier simplegit.rb est intéressant. Il vous montre que quelques lignes sont en attente et d'autres non. Vous avez mis partiellement ce fichier en attente. Dès lors, vous pouvez quitter l'ajout interactif et exécuter `git commit` pour commiter les fichiers partiellement en attente.

Enfin, vous pouvez vous passer du mode interactif pour mettre partiellement un fichier en attente; vous pouvez faire de même avec `git add -p` ou `git add --patch` en ligne de commande.

6.3 La remise

Souvent, lorsque vous avez travaillé sur une partie de votre projet, les choses sont dans un état instable mais vous voulez changer de branches pour un peu de travailler sur autre chose. Le problème est que vous ne voulez pas consigner (commit) un travail à moitié fait seulement pour pouvoir y revenir plus tard. La réponse à cette problématique est la commande `git stash`.

Remiser prend l'état en cours de votre répertoire de travail, c'est-à-dire les fichiers modifiés et la zone d'attente, et l'enregistre dans la pile des modifications non finies que vous pouvez réappliquer à n'importe quel moment.

6.3.1 Remiser votre travail

Pour démontrer cette possibilité, vous allez dans votre projet et commencez à travailler sur quelques fichiers et mettre en zone d'attente l'un de ces changements. Si vous exécutez `git status`, vous pouvez voir votre état instable:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

À ce moment là, vous voulez changer de branche, mais vous ne voulez pas encore consigner ce travail; vous allez donc remiser vos modifications. Pour créer une nouvelle remise sur votre pile, exécutez `git stash` :

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Votre répertoire de travail est propre :

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

À ce moment, vous pouvez facilement changer de branche et travailler autre part; vos modifications sont conservées dans votre pile. Pour voir quelles remises vous avez sauvegardées, vous pouvez utiliser la commande `git stash list` :

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

Dans ce cas, deux remises on été créées précédemment, vous avez donc accès à trois travaux remisés différents. Vous pouvez réappliquer celui que vous venez juste de remisé en utilisant la

commande affichée dans la sortie d'aide de la première commande de remise : `git stash apply`. Si vous voulez appliquer une remise plus ancienne, vous pouvez la spécifier en la nommant, comme ceci : `git stash apply stash@2`. Si vous ne spécifier pas une remise, Git présume que vous voulez la remise la plus récente et essayes de l'appliquer.

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

Vous pouvez observer que Git remodifie les fichiers non consignés lorsque vous avez créé la remise. Dans ce cas, vous aviez un répertoire de travail propre lorsque vous avez essayer d'appliquer la remise, et vous l'avez fait sur la même branche que celle où vous l'aviez créée; mais avoir un répertoire de travail propre et l'appliquer sur la même branche n'est pas nécessaire pour réussir à appliquer une remise. Vous pouvez très bien créer une remise sur une branche, changer de branche et essayer d'appliquer les modifications. Vous pouvez même avoir des fichiers modifiés et non consignés dans votre répertoire de travail quand vous appliquez une remise, Git vous fournit les conflits de fusions si quoique ce soit ne s'applique pas proprement.

Par défaut, les modifications de vos fichiers sont réappliqués, mais pas les mises en attente. Pour cela, vous devez exécutez la commande `git stash apply` avec l'option `--index` pour demandez à Git d'essayer de réappliquer les modifications de votre zone d'attente. Si vous exécutez cela à la place de la commande précédente, vous vous retrouvez dans la position d'origine de la remise :

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

L'option `apply` essaye seulement d'appliquer le travail remisé, vous aurez toujours la remise dans votre pile. Pour la supprimer, vous pouvez exécuter `git stash drop` avec le nom de la remise à supprimer :

```
$ git stash list
```

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Vous pouvez également exécutez `git stash pop` pour appliquer et supprimer immédiatement la remise de votre pile.

6.3.2 Créer une branche depuis une remise

Si vous remiser votre travail, l'oubliez pendant un temps en continuant sur la branche où vous avez créé la remise, vous pouvez avoir un problème en réappliquant le travail. Si l'application de la remise essaye de modifier un fichier que vous avez modifié depuis, vous allez obtenir des conflits de fusion et vous devrez essayer de les résoudre. Si vous voulez un moyen plus facile de tester une nouvelle fois les modifications remisées, vous pouvez exécuter `git stash branch`, qui créera une nouvelle branche à votre place, récupérant le commit où vous étiez lorsque vous avez créé la remise, réappliquera votre travail dedans, et supprimera finalement votre remise si cela a réussi :

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

C'est un bon raccourci pour récupérer du travail remisé facilement et de pouvoir travailler dessus dans une nouvelle branche.

6.4 Réécrire l'historique

Bien souvent, lorsque vous travaillez avec Git, vous souhaitez modifier votre historique de consignation pour une raison quelconque. Une des choses merveilleuses de Git est qu'il vous permet de prendre des décisions le plus tard possible. Vous pouvez décider quels fichiers vont dans quel commit avant que vous ne consigniez la zone d'attente, vous pouvez décider que vous ne voulez pas encore montrer que vous travailler sur quelque chose avec les remises, et vous pouvez réécrire les commits afin déjà sauvegardé pour qu'ils ressemblent à quelque chose d'autre. Cela peut signifier changer l'ordre des commits, modifier les messages ou modifier les fichiers appartenant au commit,

rassembler ou séparer des commits, ou supprimer complètement des commits; tout ceci avant de les partager avec les autres.

Danc cette section, nous expliquerons comment accomplir ces tâches très utiles pour que vous pussiez faire ressembler votre historique de consignation de la manière que vous voulez avant de le partager avec autrui.

6.4.1 Modifier la dernière consignation

Modifier votre dernière consignation est probablement la plus habituelle réécriture de l'historique que vous allez faire. Vous voudrez souvent faire deux choses basiques à votre dernier commit : modifier le message de consignation, ou changer le contenu que vous avez enregistré en ajoutant, modifiant ou supprimant des fichiers.

Si vous voulez seulement modifier votre dernier message de consignation, c'est vraiment simple :

```
$ git commit --amend
```

Cela vous ouvre votre éditeur de texte contenant votre dernier message, prêt à être modifié. Lorsque vous sauvegardez et fermez l'éditeur, Git enregistre la nouvelle consignation contenant le message et en fait votre dernier commit.

Si vous avez vouler modifier le contenu de votre consignation, en ajoutant ou modifiant des fichiers, sûrement parce que vous avez oublié d'ajouter les fichiers nouvellement créés quand vous avez consigné la première fois, la procédure fonctionne grosso-modo de la même manière. Vous mettez les modifications que vous voulez en attente en exécutant `git add` ou `git rm`, et le prochain `git commit --amend` prendra votre zone d'attente courante et en fera le contenu de votre nouvelle consignation.

Vous devez être prudent avec cette technique car votre modification modifie également le SHA-1 du commit. Cela ressemble à un tout petit `rebase`. Ne modifiez pas votre dernière consignation si vous l'avez déjà publié !

6.4.2 Modifier plusieurs messages de consignation

Pour modifier une consignation qui est plus loin dans votre historique, vous devez utilisez des outils plus complexes. Git ne contient pas d'outil de modification d'historique, mais vous pouvez utiliser l'outil `rebase` pour rebaser une suite de commits depuis la branche HEAD plutôt que de les déplacer vers une autre branche. Avec l'outil rebase interactif, vous pouvez vous arrêter après chaque commit que vous voulez modifiez et changer le message, ajouter des fichiers ou quoique ce soit que vous voulez. Vous pouvez exécuter rebase interactivement en ajoutant l'option `-i` à `git rebase`. Vous devez indiquer jusqu'à quand remonter dans votre historique en donnant à la commande le commit sur lequel vous voulez vous rebaser.

Par exemple, si vous voulez modifier les 3 derniers messages de consignation, ou n'importe lequel des messages dans ce groupe, vous fournissez à `git rebase -i` le parent du dernier commit que vous voulez éditer, qui est `HEAD~2^` or `HEAD~3`. Il peut être plus facile de ce souvenir de `~3`, car vous essayer de modifier les 3 derniers commits, mais gardez à l'esprit que vous désignez le 4e, le parent du dernier commit que vous voulez modifier :

```
$ git rebase -i HEAD~3
```

Souvenez-vous également que ceci est une commande de rebasement, chaque commit include dans l'intervalle `HEAD~3..HEAD` sera réécrit, que vous changiez le message ou non. N'incluez pas dans cette commande de commit que vous avez déjà poussé sur un serveur central. Le faire entrainera la confusion chez les autres développeurs en leur fournissant une version altérée des mêmes modifications.

Exécuter cette commande vous donne la liste des consignations dans votre éditeur de texte, ce qui ressemble à :

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Il est important de signaler que les commits sont listés dans l'ordre opposé que vous voyez normalement en utilisant la commande `log`. Si vous exécutez la commande `log`, vous verrez quelque chose de ce genre :

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Remarquez l'ordre inverse. Le rebasage interactif va créer un script à exécuter. Il commencera au commit que vous spécifiez sur la ligne de commande (`HEAD~3`) et refait les modifications introduites dans chacun des commits du début à la fin. Il ordonne donc le plus vieux au début, plutôt que le plus récent, car c'est celui qu'il refera en premier.

Vous devez éditer le script afin qu'il s'arrête au commit que vous voulez modifier. Pour cela, remplacer le mot « pick » par le mot « edit » pour chaque commit après lequel vous voulez que le script s'arrête. Par exemple, pour modifier uniquement le message du troisième commit, vous modifiez le fichier pour ressembler à :

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Au moment où vous sauvegardez et quittez l'éditeur, Git revient au dernier commit de cette liste et vous laisse sur une ligne de commande avec le message suivant :

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Ces instructions vous disent exactement quoi faire. Entrez :

```
$ git commit --amend
```

Modifiez le message de commit et quittez l'éditeur. Puis exécutez :

```
$ git rebase --continue
```

Cette commande appliquera les deux autres commits automatiquement, c'est fait. Si vous remplacez « pick » en « edit » sur plusieurs lignes, vous pouvez répéter ces étapes pour chaque commit que vous avez remplacé pour modification. Chaque fois, Git s'arrêtera, vous laissant modifier le commit et continuera lorsque vous aurez fini.

6.4.3 Réordonner les commits

Vous pouvez également utilisez les rebasages interactifs afin de réordonner ou supprimer entièrement des commits. Si vous voulez supprimer le commit « added cat-file » et modifier l'ordre dans lequel les deux autres commits se trouvent dans l'historique, vous pouvez modifier le script de rebasage :

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

afin qu'il ressemble à ceci :

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Lorsque vous sauvegardez et quittez l'éditeur, Git remet votre branche au niveau du parent de ces commits, applique `310154e` puis `f7f3f6d` et s'arrête. Vous venez de modifier l'ordre de ces commits et de supprimer entièrement le commit « added cat-file ».

6.4.4 Rassembler des commits

Il est également possible de prendre une série de commits et de les rassembler en un seul avec l'outil de rebasage interactif. Le script affiche des instructions utiles dans le message de rebasage :

```
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Si, à la place de « pick » ou « edit », vous spécifiez « squash », Git applique cette modification et la modification juste précédente et fusionne les messages de consignation. Donc, si vous voulez faire un seul commit de ces trois consignations, vous faites en sorte que le script ressemble à ceci :

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Lorsque vous sauvegardez et quittez l'éditeur, Git applique ces trois modifications et vous remontre l'éditeur contenant maintenant la fusion des 3 messages de consignation :

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

Lorsque vous sauvegardez cela, vous obtenez un seul commit amenant les modifications des trois commits précédents.

6.4.5 Diviser un commit

Pour diviser un commit, il doit être défait, puis partiellement mis en zone d'attente et consigner autant de fois que vous voulez pour en finir avec lui. Par exemple, supposons que vous voulez diviser le commit du milieu dans l'exemple des trois commits précédents. Plutôt que « updated README formatting and added blame », vous voulez le diviser en deux commits : « updated README formatting » pour le premier, et « added blame » pour le deuxième. Vous pouvez le faire avec le script `rebase -i` en remplaçant l'instruction sur le commit que vous voulez divisez en « edit » :

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Puis, lorsque le script vous laissera accès à la ligne de commande, vous annulerez (reset) ce commit, vous reprendrez les modifications que vous voulez pour créer plusieurs commits. En reprenant l'exemple, lorsque vous sauvegardez et quittez l'éditeur, Git revient au parent de votre premier commit de votre liste, applique le premier commit (`f7f3f6d`), applique le deuxième (`310154e`), et vous laisse accès à la console. Là, vous pouvez faire une réinitialisation mélangée (mixed reset) de ce commit avec `git reset HEAD^`, qui défait ce commit et laisse les fichiers modifiés non mis en attente. Maintenant, vous pouvez mettre en attente et consigner les fichiers sur plusieurs consignations, et exécuter `git rebase --continue` quand vous avez fini :

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applique le dernier commit (`a5f4a0d`) de votre script, et votre historique ressemblera alors à :

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Une fois encore, ceci modifie les empreintes SHA de tous les commits dans votre liste, soyez donc sûr qu'aucun commit de cette liste ait été poussé dans un dépôt partagé.

6.4.6 L'option nucléaire : filter-branch

Il existe une autre option de la réécriture d'historique que vous pouvez utiliser si vous avez besoin de réécrire un grand nombre de commits d'une manière scriptable; par exemple, modifier globalement votre adresse mail ou supprimer un fichier de tous les commits. La commande est `filter-branch`, et elle peut réécrire des pans entiers de votre historique, vous ne devriez donc pas l'utiliser à moins que votre projet ne soit pas encore public ou que personne n'a encore travaillé sur les commits que vous allez réécrire. Cependant, cela peut être très utile. Vous allez maintenant apprendre quelques usages communs pour vous donner une idée de ses capacités.

Supprimer un fichier de chaque commit

Cela arrive asser fréquemment. Quelqu'un a accidentellement commité un énorme fichier binaire avec une commande `git add .` irréfléchie, and vous voulez le supprimer partout. Vous avez peut-être consigné un fichier contenant un mot de passe, et que vous voulez rendre votre projet open source. `filter-branch` est l'outil que vous voulez probablement utiliser pour nettoyer votre historique entier. Pour supprimer un fichier nommé « passwords.txt » de tout votre historique, vous pouvez utiliser l'option `--tree-filter` de `filter-branch` :

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

L'option `--tree-filter` exécute la commande spécifiée pour chaque commit et les reconsigne ensuite Dans le cas présent, vous supprimez le fichier nommé « passwords.txt » de chaque contenu, qu'il existait ou non. Si vous voulez supprimez tous les fichiers temporaires des éditeurs consignés accidentellement, vous pouvez exécuter une commande telle que `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Vous pourrez alors regarder Git réécrire l'arbre des commits et reconsigner à chaque fois, pour finir en modifiant la référence de la branche. C'est généralement une bonne idée de le faire dans un branche de test puis de faire une forte réinitialisation (hard-reset) de votre branche `master` si le résultat vous convient. Pour exécuter `filter-branch` sur toutes vos branches, vous pouvez ajouter `--all` à la commande.

Faire d'un sous-répertoire la nouvelle racine

Supposons que vous avez importer votre projet depuis un autre système de gestion de configuration et que vous avez des sous-répertoires qui n'ont aucun sens (trunk, tags, etc). Si vous voulez faire en sorte que le sous-répertoire `trunk` soit la nouvelle racine de votre projet pour tous les commits, `filter-branch` peut aussi vous aider à le faire :

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Maintenant votre nouvelle racine est remplacé par le contenu du répertoire `trunk`. De plus, Git supprimera automatiquement les commits qui n'affectent pas ce sous-répertoire.

Modifier globalement l'adresse mail

Un autre cas habituel est que vous oubliez d'exécuter `git config` pour configurer votre nom et votre adresse mail avant de commencer à travailler, ou vous voulez peut-être rendre un projet du boulot open source et donc changer votre adresse professionnelle pour celle personnelle. Dans tous les cas, vous pouvez modifier l'adresse mail dans plusieurs commits avec un script `filter-branch` Vous devez faire attention de ne changer que votre adresse mail, utilisez donc `--commit-filter` :

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

Cela passe sur chaque commit et le réécrit pour avoir votre nouvelle adresse. Mais puisque les commits contiennent l'empreinte SHA-1 de leur parent, cette commande modifie tous les commits dans votre historique, pas seulement ceux correspondant à votre adresse mail.

6.5 Deboguer avec Git

Git fournit aussi quelques outils pour vous aider à déboguer votre projet. Puisque Git est conçu pour fonctionner avec pratiquement tout type de projet, ces outils sont plutôt génériques, mais ils peuvent souvent vous aider à traquer un bogue ou au moins cerner où cela tourne mal.

6.5.1 Fichier annoté

Si vous traquez un bogue dans votre code et que vous voulez savoir quand il est apparu et pourquoi, annoter les fichiers est souvent le meilleur moyen. Cela vous montre le dernier commit qui a modifié chaque ligne de votre fichier. Donc, si vous voyez une méthode dans votre code qui est boguée, vous pouvez annoter le fichier avec `git blame` pour voir quand chaque ligne de la méthode a été modifiée pour la dernière fois et par qui. Cet exemple utilise l'option `-L` pour limiter la sortie des lignes 12 à 22 :

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
```

```
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Remarquez que le premier champ est le SHA-1 partiel du dernier commit a avoir modifié la ligne. Les deux champs suivants sont des valeurs extraites du commit : l'auteur et la date du commit, vous pouvez donc facilement voir qui a modifié la ligne et quand. Ensuite arrive le numéro de ligne et son contenu. Remarquez également les lignes dont le commit est `^4832fe2`, elles désignent les lignes qui étaient dans la version du fichier lors du premier commit de ce fichier. Ce commit contient le premier ajout de ce fichier, et ces lignes n'ont pas été modifiées depuis. Tout ça est un peu confus, parce que vous connaissez maintenant au moins trois façons différentes que Git interprète `^` pour modifier l'empreinte SHA, mais au moins, vous savez ce qu'il signifie ici.

Une autre chose sympa sur Git, c'est qu'il ne suit pas explicitement les renommages de fichier. Il enregistre les contenus puis essaye de deviner ce qui a été renommé implicitement, après coup. Ce qui nous permet d'utiliser cette fonctionnalité intéressante pour suivre toute sorte de mouvement de code. Si vous passez `-C` à `git blame`, Git analyse le fichier que vous voulez annoter et essaye de devenir d'où les bouts de code proviennent par copie ou déplacement. Récemment, j'ai remanié un fichier nommé `GITServerHandler.m` en le divisant en plusieurs fichiers, dont le fichier `GITPackUpload.m`. En annotant `GITPackUpload.m` avec l'option `-C`, je peux voir quelles sections de code en est originaire :

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

C'est vraiment utile, non ? Normalement, vous obtenez comme commit original, celui dont votre code a été copié, puisque ce fut la première fois que vous avez touché à ces lignes dans ce fichier. Git vous montre le commit d'origine, celui où vous avez écrit ces lignes, même si c'était dans un autre fichier.

6.5.2 La recherche dichotomique

Annoter un fichier peut aider si vous savez déjà où le problème se situe. Si vous ne savez pas ce qui a cassé le code, il peut y avoir des douzaines, voire des centaines de commits depuis le dernier état où votre code fonctionnait, et vous aimeriez certainement exécuter `git bisect` pour l'aide qu'il procure. La commande `bisect` effectue une recherche par dichotomie dans votre historique pour vous aider à identifier aussi vite que possible quel commit a vu le bogue naitre.

Disons que vous venez juste de pousser une version finale de votre code en production, vous récupérez un rapport de bogue à propos de quelque chose qui n'arrivait pas dans votre environnement de développement, et vous n'arrivez pas à trouver pourquoi votre code le fait. Vous retournez sur votre code, et il apparait que vous pouvez reproduire le bogue, mais vous ne savez pas ce qui se passe mal. Vous pouvez faire une recherche par dichotomie pour trouver ce qui ne va pas. D'abord, exécutez `git bisect start` pour démarrer la procédure, puis utilisez la commande `git bisect bad` pour dire que le commit courant est bogué. Ensuite, dites à `bisect` quand le code fonctionnait, en utilisant `git bisect good [bonne_version]` :

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git trouve qu'il y a environ 12 commits entre celui que vous avez marqué comme le dernier bon connu (v1.0) et la version courante qui n'est pas bonne, et il a récupéré le commit au milieu à votre place. À ce moment, vous pouvez dérouler vos tests pour voir si le bogue existait dans ce commit. Si c'est le cas, il a été introduit quelque part avant ce commit médian, sinon, il l'a été évidemment introduit après. Il apparait que le bogue ne se reproduit pas ici, vous le dites à Git en tapant `git bisect good` et continuez votre périple :

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Vous êtes maintenant sur un autre commit, à mi-chemin entre celui que vous venez de tester et votre commit bogué. Vous exécutez une nouvelle fois votre test et trouvez que ce commit est bogué, vous le dites à Git avec `git bisect bad` :

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Ce commit-ci est bon, et Git a maintenant toutes les informations dont il a besoin pour déterminer où le bogue a été créé. Il vous affiche le SHA-1 du premier commit bogué, quelques informations du commit et quels fichiers ont été modifiés dans celui-ci, vous pouvez donc trouver ce qui s'est passé pour créer ce bogue :

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
```

```
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Lorsque vous avez fini, vous devez exécuter `git bisect reset` pour réinitialiser votre HEAD où vous étiez avant de commencer, ou vous travaillerez dans un répertoire de travail non clairement défini :

```
$ git bisect reset
```

C'est un outil puissant qui vous aidera à vérifier des centaines de commits en quelques minutes. En réalité, si vous avez un script qui sort avec une valeur `0` s'il est bon et autre chose sinon, vous pouvez même automatiser `git bisect`. Premièrement vous lui spécifiez l'intervalle en lui fournissant les bon et mauvais commits connus. Vous pouvez faire cela en une ligne en les entrant à la suite de la commande `bisect start`, le mauvais commit d'abord :

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Cela exécute automatiquement `test-error.sh` sur chaque commit jusqu'à ce que Git trouve le premier commit bogué. Vous pouvez également exécuter des commandes comme `make` ou `make tests` ou quoique ce soit qui exécute des tests automatisés à votre place.

6.6 Sous-modules

Il arrive souvent lorsque vous travaillez sur un projet, que vous devez utilisez un autre projet comme dépendance. Cela peut être une librairie qui est développée par une autre équipe ou que vous développez séparemment pour l'utiliser dans plusieurs projets parents. Ce scénario provoque un problème habituel : vous voulez être capable de gérer deux projets séparés tout en utilisant un dans l'autre.

Voici un exemple. Supposons que vous développez un site web et que vous créez des flux Atom. Plutôt que d'écrire votre propre code de génération Atom, vous décidez d'utiliser une librairie. Vous allez vraisemblablement devoir soit inclure ce code depuis un gestionnaire partagé comme CPAN ou Ruby gem, soit copier le code source dans votre propre arborescence de projet. Le problème d'inclure la librairie en tant que librairie externe est qu'il est difficile de la personnaliser de quelque manière que ce soit et encore plus de la déployer, car vous devez vous assurer de la disponibilité de la librairie chez chaque client. Mais le problème d'inclure le code dans votre propre projet est que n'importe quelle personnalisation que vous faites est difficile à fusionner lorsque les modifications du développement principal arrivent.

Git gère ce problème avec les sous-modules. Les sous-modules vous permettent de gérer un dépôt Git comme un sous-répertoire d'un autre dépôt Git. Cela vous laisse la possibilité de cloner un dépôt dans votre projet et de garder isolés les commits de ce dépôt.

6.6.1 Démarrer un sous-module

Supposons que vous voulez ajouter la librairie Rack (un serveur d'application web en Ruby) à votre projet, avec la possibilité de gérer vos propres changements à celle-ci mais en continuant de fusionner avec la branche principale. La première chose que vous devez faire est de cloner le dépôt externe dans votre sous-répertoire. Ajouter des projets externes comme sous-modules de votre projet se fait avec la commande `git submodule add` :

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Vous avez maintenant le projet Rack dans le sous-répertoire `rack` à l'intérieur de votre propre projet. Vous pouvez aller dans ce sous-répertoire, effectuer des modifications, ajouter votre propre dépôt distant pour y pousser vos modifications, récupérer et fusionner depuis le dépôt originel, et plus encore. Si vous exécutez `git status` juste après avoir ajouter le sous-module (donc dans le répertoire parent du répertoire `rack`), vous verrez deux choses :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      new file:   .gitmodules
#      new file:   rack
#
```

Premièrement, vous remarquerez le fichier `.gitmodules`. C'est un fichier de configuration sauvegardant la liaison entre l'URL du projet et le sous-répertoire local où vous l'avez mis :

```
$ cat .gitmodules
[submodule "rack"]
      path = rack
      url = git://github.com/chneukirchen/rack.git
```

Si vous avez plusieurs sous-modules, vous aurez plusieurs entrées dans ce fichier. Il est important de noter que ce fichier est en gestion de version comme vos autres fichiers, à l'instar de votre fichier `.gitignore`. Il est poussé et tiré comme le reste de votre projet. C'est également le

moyen que les autres personnes qui clonent votre projet peuvent savoir où récupérer le projet du sous-module.

L'autre information dans la sortie de `git status` est l'entrée `rack`. Si vous exécutez `git diff`, vous verrez quelque chose d'intéressant :

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Même si `rack` est un sous répertoire de votre répertoire de travail, Git le voit comme un sous-module et ne suit pas son contenu (si vous n'êtes pas dans ce répertoire). En échange, Git l'enregistre comme un commit particulier de ce dépôt. Lorsque vous faites des modifications et des consignations dans ce sous-répertoire, le super-projet (le projet contenant le sous-module) remarque que la branche HEAD a changé et enregistre le commit exacte dans lequel il se trouve à ce moment. De cette manière, lorsque d'autre clone ce super-projet, ils peuvent recréer exactement le même environnement.

Un autre point important avec les sous-modules : Git enregistre le commit exact où ils se trouvent. Vous ne pouvez pas enregistrer un module comme étant en branche `master` ou n'importe quelle autre référence symbolique.

Au moment de commiter, vous voyez quelque chose comme :

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

Remarquez le mode 160000 pour l'entrée `rack`. C'est un mode spécial de Git qui signifie globalement que vous êtes en train d'enregistrer un commit comme un répertoire plutôt qu'un sous-répertoire ou un fichier.

Vous pouvez traiter le répertoire `rack` comme un projet séparé et mettre à jour votre super-projet de temps en temps avec une référence au dernier commit de ce sous-projet. Toutes les commandes Git fonctionnent indépendamment dans les deux répertoires :

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack
$ cd rack/
```

```
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

6.6.2 Cloner un projet avec des sous-modules

Maintenant, vous allez apprendre à cloner un projet contenant des sous-modules. Quand vous récupérez un tel projet, vous obtenez les différents répertoires qui contiennent les sous-modules, mais encore aucun des fichiers :

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r-r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
$
```

Le répertoire `rack` est présent mais vide. Vous devez exécuter deux commandes : `git submodule init` pour initialiser votre fichier local de configuration, et `git submodule update` pour tirer toutes les données de ce projet et récupérer le commit approprié tel que listé dans votre super-projet :

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

Votre répertoire `rack` est maintenant dans l'état exacte dans lequel il était la dernière fois que vous avez consigné. Si un autre développeur modifie le code de `rack` et consigne, que vous récupériez (pull) cette référence et que vous fusionniez, vous obtiendrez quelque chose d'un peu étrange :

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#      modified:   rack
#
```

En réalité, vous n'avez fusionné que la modification de la référence de votre sous-module, mais Git n'a pas mis à jour le code dans le répertoire du sous-module, de ce fait, cela ressemble à un état « en cours » dans votre répertoire de travail :

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

La cause de tout cela, c'est que la référence pour votre sous-module ne correspond pas à ce qu'il y a actuellement dans son répertoire. Pour corriger ça, vous devez exécuter un nouvelle fois `git submodule update` :

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master     -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

Vous devez faire cela à chaque fois que vous récupérez une modification du sous-module dans le projet principal. C'est étrange, mais ça fonctionne.

Un problème habituel peut survenir lorsqu'un développeur modifie localement un sous-module, mais ne le pousse pas sur un serveur public. Puis, il consigne une référence à cet état non public et pousse le super-projet. Lorsque les autres développeurs exécutent `git submodule update`, le système dans le sous-module ne trouve pas le commit qui est référencé, car il existe uniquement sur le système du premier développeur. Dans ce cas, vous verrez une erreur de ce style :

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

Vous devez regarder qui a modifié le sous-module en dernier :

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

Envoyez-lui un mail pour lui gueuler dessus.

6.6.3 Superprojects

Parfois, les développeurs désirent séparer un gros projet en sous-répertoires en fonction de l'équipe qui travaille dessus. C'est logique que si vous venez de CVS ou de Subversion, où vous aviez l'habitude de définir un module ou un ensemble de sous-répertoires, que vous vouliez garder ce type de workflow.

Une bonne manière de le faire avec Git est de créer un dépôt Git pour chaque sous-dossiers, et de créer un super-projet contenant les différents modules. Le bénéfice de cette approche est de pouvoir spécifier les relations entre les projets avec des étiquettes et des branches depuis le super-projet.

6.6.4 Les problèmes avec les sous-modules

Cependant, utiliser des sous-modules ne se déroule pas sans accroc. Premièrement, vous devez être relativement prudent lorsque vous travaillez dans le répertoire du sous-module. Lorsque vous exécutez `git submodule update`, cela récupère une version spécifique d'un projet, mais pas à l'intérieur d'une branche. Cela s'appelle avoir la tête en l'air (detached head), c'est-à-dire que votre HEAD référence directement un commit, pas une référence symbolique. Le problème est que vous ne voulez généralement pas travailler dans un environnement tête en l'air, car il est facile de perdre des modifications dans ces conditions. Si vous faites un premier `git submodule update`, consignez des modifications dans ce sous-module sans créer vous-même de branche pour y travailler, et que vous exécutez un nouveau `git submodule update` depuis le projet parent sans y avoir consigné pendant ce temps, Git écrasera vos modifications sans vous le dire. Techniquement, vous ne perdrez pas votre travail, mais vous n'aurez aucune branche s'y référant, il sera donc assez difficile de le récupérer.

Pour éviter ce problème, créez toujours une branche lorsque vous travaillez dans un répertoire de sous-module avec `git checkout -b work` ou une autre commande équivalente. Lorsque vous mettrez à jour le sous-module une deuxième fois, Git réinitialisera toujours votre travail, mais vous aurez au moins une référence pour y retourner.

Commuter de branches qui contiennent des sous-modules peut également s'avérer difficile. Si vous créez une nouvelle branche, y ajoutez un sous-module, et revenez ensuite à une branche sans ce sous-module, vous aurez toujours le répertoire de ce sous-module comme un répertoire non suivi :

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      rack/
```

Vous devez soit déplacer ce répertoire hors de votre dépôt local, soit le supprimer, dans ce dernier ca, vous devrait le clôner une nouvelle fois lorsque vous recommuterez et vous pouvez donc perdre des modifications ou des branches locales si vous ne les avez pas poussées.

Le dernier piège dans lequel beaucoup tombe est de passer des sous-répertoires à des sous-modules. Si vous suiviez des fichiers dans votre projet et que vous voulez les déplacer dans un sous-module, vous devez être très prudent, ou Git vous mangera. Présumons que vous avez des fichiers rack dans un sous-répertoire de votre projet, et que vous voulez les transformer en un sous-module. Si vous supprimez le sous-répertoire et que vous exécutez `submodule add`, Git vous hurle dessus avec :

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

Vous devez d'abord supprimer le répertoire `rack` de la zone d'attente. Vous pourrez ensuite ajouter le sous-module :

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
```

```
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

Maintenant, supposons que vous avez fait cela dans une branche. Si vous essayer de commuter dans une ancienne branche où ces fichiers sont toujours dans l'arbre de projet plutôt que comme sous-module, vous aurez cette erreur :

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Vous devez déplacer le répertoire du sous-module `rack` en dehors de votre dépôt local avant de pouvoir commuter vers une branche qui ne l'a pas :

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

Puis, lorsque vous recommutez, vous aurez un répertoire `rack` vide. Vous pouvez soit exécuter `git submodule update` pour clôner une nouvelle fois, ou vous pouvez remettre votre répertoire `/tmp/rack` dans votre répertoire vide.

6.7 Fusion de sous-arborescence

Maintenant que vous avez vu les difficultés qu'il peut y avoir avec le système de sous-module, voyons une alternative pour résoudre la même problématique. Lorsque Git fusionne, il regarde ce qu'il doit fusionner et choisit alors une stratégie de fusion appropriée. Si vous fusionnez deux branches, Git utilise une stratégie *récursive* (*recursive* strategy). Si vous fusionnez plus de deux branches, Git choisit la stratégie de la *pieuvre* (*octopus* strategy). Ces stratégies sont choisies automatiquement car la stratégie récursive peut gérer des problèmes comples de fusions à trois entrées, par exemple, plus d'un ancêtre commun, mais il ne peut gérer que deux branches. La fusion de la pieuvre peut gérer plusieurs branches mais il est plus prudent afin d'éviter les conflits difficiles, il est donc choisi comme stratégie par défaut si vous essayez de fusionner plus de deux branches.

Cependant, il existe d'autres stratégies que vous pouvez tout aussi bien choisir. L'une d'elles est la fusion de sous-arborescence, et vous pouvez l'utiliser pour gérer la problématique de sous-projet. Nous allons donc voir comme gérer l'inclusion de rack comme dans la section précédente, mais en utilisant cette fois-ci les fusion de sous-arborescence.

La fusion de sous-arborescence suppose que vous avez deux projets et que l'un s'identifie à un sous-répertoire de l'autre. Lorsque vous spécifiez une fusion de sous-arborescence, Git est assez intellignet pour deviner lequel est un sous-répertoire de l'autre et fusionne en conséquence, When

you specify a subtree merge, Git is smart enough to figure out that one is a subtree of the other and merge appropriately — c'est assez bluffant.

Premièrement, vous ajoutez l'application Rack à votre projet. Vous ajoutez le projet Rack comme une référence distante dans votre propre projet et récupérez dans un branche personnelle :

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Vous avez maintenant la racine du projet Rack dans votre branche `rack_branch` et votre propre projet dans la branche `master`. Si vous récupérez l'une puis l'autre branche, vous pouvez voir que vous avez différentes racines de projet :

```
$ ls
AUTHORS        KNOWN-ISSUES   Rakefile      contrib        lib
COPYING        README         bin           example        test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Pour tirer le projet Rack dans votre projet `master` comme un sous répertoire, vous pouvez utiliser la commande `git read-tree`. Vous apprendrez d'avantage sur `read-tree` et compagnie dans le Chapitre 9, mais pour le moment, sachez qu'il lit la racine d'une de vos branche et l'inscrit dans votre zone d'attente et votre répertoire de travail. Vous venez juste de commuter vers votre branche `master`, et vous tirez la branche `rack` vers le sous-répertoire `rack` de votre branche `master` de votre projet principal :

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Au moment de consigner, vous verrez tout les fichiers de Rack de ce sous-répertoire, comme si vous les aviez copié depuis une archive. Ce qui est intéressant, c'est que vous pouvez assez facilement fusionner les changements d'une branche à l'autre. Par conséquence, s'il y a des mises à jour pour

le projet Rack, vous pouvez les tirez depuis le dépôt principal en commutant dans cette branche et tirant les modifications :

```
$ git checkout rack_branch
$ git pull
```

Puis, vous pouvez fusionner ces changements dans votre branche principale. Vous pouvez utiliser `git merge -s subtree` et cela fonctionnera, mais Git fusionnera également les historiques ensemble, ce que vous ne voulez probablement pas. Pour tirer les changements et préremplir le message de consignation, utilisez les options `--squash` et `--no-commit` avec l'option de stratégie `-s subtree` :

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Toutes les modifications de votre projet Rack sont fusionné et prêtes à être consignées localement. Vous pouvez également faire le contraire, faire des modifications dans le sous-répertoire `rack` de votre branche principale et les fusionner plus tard dans votre branche `rack_branch` pour les envoyer aux mainteneurs du projet Rack ou les pousser dans le dépôt principal.

Pour voir les différences entre ce que vous avez dans le sous-répertoire `rack` et le code de la branche `rack_branch` (pour savoir si vous devez les fusionner), vous ne pouvez pas utiliser la commande `diff` habituelle. Vous devez plutôt exécutez `git diff-tree` en renseignant la branche avec laquelle vous voulez comparer :

```
$ git diff-tree -p rack_branch
```

Ou, pour comparer ce qu'il y a dans votre répertoire `rack` avec ce qu'il y avait sur le server la dernière fois que vous avez vérifié, vous pouvez exécuter :

```
$ git diff-tree -p rack_remote/master
```

6.8 Résumé

Vous venez de voir certains des outils avancés vous permettant de manipuler vos consignations et votre zone d'attente plus précisemment. Lorsque vous remarquez des bogues, vous devriez être capable de facilement trouver quelle consignation les a introduits, quand et par qui. Si vous voulez utiliser des sous-projets dans votre projet, vous avez appris plusieurs façons de les gérer. À partir de maintenant, vous devez être capable de faire la majorité de ce que vous avez besoin avec Git en ligne de commande et de vous y sentir à l'aise.

Chapitre 7

Personnalisation de Git

Jusqu'ici, nous avons traité les bases du fonctionnement et de l'utilisation de Git et introduit un certain nombre d'outils fournis par Git pour travailler plus facilement et plus efficacement. Dans ce chapitre, nous aborderons quelques opérations permettant d'utiliser Git de manière plus personnalisée en vous présentant quelques paramètres de configuration importants et le système d'interceptions. Grâce à ces outils, il devient enfantin de faire fonctionner Git exactement comme vous, votre société ou votre communauté en avez besoin.

7.1 Configuration de Git

Comme vous avez pu l'entrevoir au chapitre 1, vous pouvez spécifier les paramètres de configuration de Git avec la commande `git config`. Une des premières choses que vous avez faites a été de paramétrer votre nom et votre adresse e-mail :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

À présent, vous allez apprendre quelques unes des options similaires les plus intéressantes pour paramétrer votre usage de Git.

Vous avez vu des détails de configuration simple de Git au premier chapitre, mais nous allons les réviser. Git utilise une série de fichiers de configuration pour déterminer son comportement selon votre personnalisation. Le premier endroit que Git visite est le fichier `/etc/gitconfig` qui contient des valeurs pour tous les utilisateurs du système et tous leurs dépôts. Si vous passez l'option `--system` à `git config`, il lit et écrit ce fichier.

L'endroit suivant visité par Git est le fichier `~/.gitconfig` qui est spécifique à chaque utilisateur. Vous pouvez faire lire et écrire Git dans ce fichier au moyen de l'option `--global`.

Enfin, Git recherche des valeurs de configuration dans le fichier de configuration du répertoire Git (`.git/config`) du dépôt en cours d'utilisation. Ces valeurs sont spécifiques à un unique dépôt. Chaque niveau surcharge le niveau précédent, ce qui signifie que les valeurs dans `.git/config` écrasent celles dans `/etc/gitconfig`. Vous pouvez positionner ces valeurs manuellement en éditant le fichier et en utilisant la syntaxe correcte, mais il reste généralement plus facile de lancer la commande `git config`.

7.1.1 Configuration de base d'un client

Les options de configuration reconnues par Git tombent dans deux catégories : côté client et côté serveur. La grande majorité se situe côté client pour coller à vos préférences personnelles de travail. Parmi les tonnes d'options disponibles, seules les plus communes ou affectant significativement la manière de travailler seront traitées. De nombreuses options ne s'avèrent utiles que sur des cas rares et ne seront pas traitées. Pour voir la liste des toutes les options que votre version de Git reconnaît, vous pouvez lancer :

```
$ git config --help
```

La page de manuel pour `git config` liste aussi les options disponibles avec un bon niveau de détail.

core.editor

Par défaut, Git utilise votre éditeur par défaut ou se replie sur l'éditeur Vi pour la création et l'édition des messages de validation et de balisage. Pour modifier ce comportement par défaut pour un autre, vous pouvez utiliser le paramètre `core.editor` :

```
$ git config --global core.editor emacs
```

Maintenant, quelque soit votre éditeur par défaut, Git démarrera Emacs pour éditer les messages.

commit.template

Si vous réglez ceci sur le chemin d'un fichier sur votre système, Git utilisera ce fichier comme message par défaut quand vous validez. Par exemple, supposons que vous créez un fichier modèle dans `$HOME/.gitmessage.txt` qui ressemble à ceci :

```
ligne de sujet

description

[ticket: X]
```

Pour indiquer à Git de l'utiliser pour le message par défaut qui apparaîtra dans votre éditeur quand vous lancerez `git commit`, réglez le paramètre de configuration `commit.template` :

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Ainsi, votre éditeur ouvrira quelque chose ressemblant à ceci comme modèle de message de validation :

```
ligne de sujet

description

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Si vous avez une règle de messages de validation, placer un modèle de cette règle sur votre système et configurer Git pour qu'il l'utiliser par défaut améliorera les chances que cette règle soit effectivement suivie.

core.pager

Le paramètre `core.pager` détermine quel *pager* est utilisé lorsque des pages de Git sont émises, par exemple lors d'un `log` ou d'un `diff`. Vous pouvez le fixer à `more` ou à votre *pager* favori (par défaut, il vaut `less`) ou vous pouvez le désactiver en fixant sa valeur à une chaîne vide :

```
$ git config --global core.pager ''
```

si vous lancez cela, Git affichera la totalité du résultat de toutes les commandes d'une traite, quelle que soit sa longueur.

user.signingkey

Si vous faîtes des balises annotées signées (comme décrit au chapitre 2), simplifiez-vous la vie en définissant votre clé GPG de signature en paramètre de configuration. Définissez votre ID de clé ainsi :

```
$ git config --global user.signingkey <gpg-key-id>
```

Maintenant, vous pouvez signer vos balises sans devoir spécifier votre clé à chaque fois à la commande `git tag` :

```
$ git tag -s <nom-balise>
```

core.excludesfile

Comme décrit au chapitre 2, vous pouvez ajouter des patrons dans le fichier `.gitignore` de votre projet pour indiquer à Git de ne pas considérer certains fichiers comme non suivis ou les indexer lorsque vous lancez `git add` sur eux. Cependant, si vous souhaitez qu'un autre fichier à l'extérieur du projet contiennent ces informations ou d'autres supplémentaires, vous pouvez indiquer à Git où se trouve ce fichier grâce au paramètre `core.excludesfile`. Fixez le simplement sur le chemin du fichier qui contient les informations similaires à celles de `.gitignore`.

help.autocorrect

Cette option n'est disponible qu'à partir de la version 1.6.1. Si vous avez fait une faute de frappe en tapant une commande dans Git 1.6, il vous affichera une liste de commandes ressemblantes :

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
     commit
```

Si vous positionnez le paramètre `help.autocorrect` à 1, Git lancera automatiquement de lui-même la commande si une seule commande ressemblante a été trouvée.

7.1.2 Couleurs dans Git

Git peut coloriser ses affichages dans votre terminal, ce qui peut faciliter le parcours visuel des résultats. Un certain nombre d'options peuvent vous aider à régler la colorisation à votre goût.

color.ui

Git colorise automatiquement la plupart de ses affichages si vous le lui demandez. Vous pouvez néanmoins vouloir être plus précis sur ce que vous souhaitez voir colorisé et comment vous le souhaitez. Pour activer toute la colorisation par défaut, fixez `color.ui` à `true` :

```
$ git config --global color.ui true
```

Avec cette valeur du paramètre, Git colorise sa sortie si celle-ci est destinée à un terminal. D'autres réglages possibles sont `false` qui désactive complètement la colorisation et `always` qui active la colorisation, même si vous envoyez la commande Git dans un fichier ou l'entrée d'une autre commande. Ce réglage a été ajouté dans Git 1.5.5. Si vous avez une version antérieure, vous devrez spécifier les règles de colorisation individuellement.

`color.ui = always` est rarement utile. Dans les plupart des cas, si vous tenez vraiment à coloriser vos sorties redirigées, vous pourrez passer le drapeau `--color` à la commande Git pour la forcer à utiliser les codes de couleur. Le réglage `color.ui = true` est donc le plus utilisé.

color.*

Si vous souhaitez être plus spécifique concernant les commandes colorisées ou si vous avez une ancienne version, Git propose des paramètres de colorisation par action. Chacun peut être fixé à `true`, `false` ou `always`.

```
color.branch
color.diff
color.interactive
color.status
```

De plus, chacun d'entre eux dispose d'un sous-ensemble de paramètres qui permettent de surcharger les couleurs pour des parties des affichages. Par exemple, pour régler les couleurs de méta-informations du diff avec une écriture en bleu gras (*bold* en anglais) sur fond noir :

```
$ git config --global color.diff.meta "blue black bold"
```

La couleur peut prendre les valeurs suivantes : *normal*, *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan* ou *white*. Si vous souhaitez ajouter un attribut de casse, les valeurs disponibles sont *bold* (gras), *dim* (léger), *ul* (*underlined*, souligné), *blink* (clignotant) et *reverse* (inversé).

Référez-vous à la page de manuel de `git config` pour tous les sous-réglages disponibles.

7.1.3 Outils externes de fusion et de différence

Bien que Git ait une implémentation interne de diff que vous avez déjà utilisée, vous pouvez sélectionner à la place un outil externe. Vous pouvez aussi sélectionner un outil graphique pour la fusion et la résolution de conflit au lieu de devoir résoudre les conflits manuellement. Je démontrerai le paramétrage avec Perforce Merge Tool (P4Merge) pour visualiser vos différences et résoudre vos fusions parce que c'est un outil graphique agréable et gratuit.

Si vous voulez l'essayer, P4Merge fonctionne sur tous les systèmes d'exploitation principaux. Dans cet exemple, je vais utiliser la forme des chemins usitée sur Mac et Linux. Pour Windows, vous devrez changer `/usr/local/bin` pour le chemin d'exécution dans votre environnement.

Vous pouvez télécharger P4Merge ici :

http://www.perforce.com/perforce/downloads/component.html

Pour commencer, créez un script d'appel externe pour lancer vos commandes. Je vais utiliser le chemin Mac pour l'exécutable ; dans d'autres systèmes, il résidera où votre binaire `p4merge` a été installé. Créez un script enveloppe nommé `extMerge` qui appelle votre binaire avec tous les arguments fournis :

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

L'enveloppe diff s'assure que sept arguments ont été fournis et en passe deux à votre script de fusion. Par défaut, Git passe au programme de diff les arguments suivants :

```
chemin ancien-fichier ancien-hex ancien-mode nouveau-fichier nouveau-hex nouveau-
mode
```

Comme seuls les arguments `ancien-fichier` et `nouveau-fichier` sont nécessaires, vous utilisez le script d'enveloppe pour passer ceux dont vous avez besoin.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Vous devez aussi vous assurer que ces outils sont exécutables :

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

À présent, vous pouvez régler votre fichier de configuration pour utiliser vos outils personnalisés de résolution de fusion et de différence. Pour cela, il faut un certain nombre de personnalisations : `merge.tool` pour indiquer à Git quelle stratégie utiliser, `mergetool.*.cmd` pour spécifier comment lancer cette commande, `mergetool.trustExitCode` pour indiquer à Git si le code de sortie du programme indique une résolution de fusion réussie ou non et `diff.external` pour indiquer à Git quelle commande lancer pour les différences. Ainsi, vous pouvez lancer les quatre commandes

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

ou vous pouvez éditer votre fichier `~/.gitconfig` pour y ajouter ces lignes :

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
```

```
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Après avoir réglé tout ceci, si vous lancez des commandes de diff telles que celle-ci :

```
$ git diff 32d1776b1^ 32d1776b1
```

Au lieu d'obtenir la sortie du diff dans le terminal, Git lance P4Merge, ce qui ressemble à la Figure 7.1.

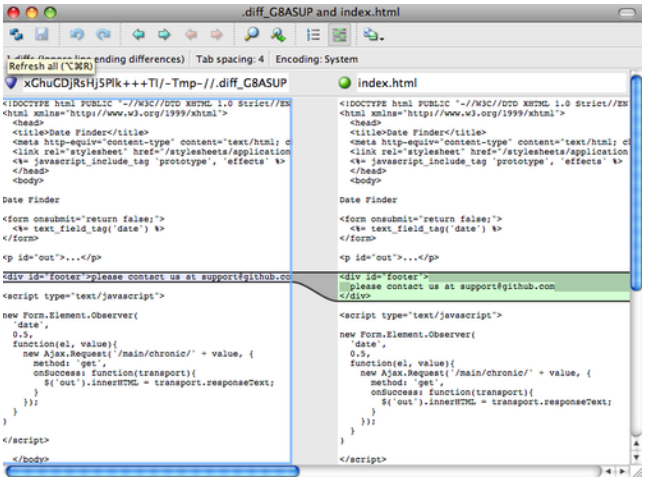

Figure 7.1: *P4Merge.*

Si vous essayez de fusionner deux branches et créez des conflits de fusion, vous pouvez lancer la commande `git mergetool` qui démarrera P4Merge pour vous laisser résoudre les conflits au moyen d'un outil graphique.

Le point agréable avec cette méthode d'enveloppe est que vous pouvez changer facilement d'outils de diff et de fusion. Par exemple, pour changer vos outils `extDiff` et `extMerge` pour une utilisation de l'outil KDiff3, il vous suffit d'éditer le fichier `extMerge` :

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

À présent, Git va utiliser l'outil KDiff3 pour visualiser les différences et résoudre les conflits de fusion.

Git est livré préréglé avec un certain nombre d'autres outils de résolution de fusion pour vous éviter d'avoir à gérer la configuration cmd. Vous pouvez sélectionner votre outil de fusion parmi `kdiff3`, `opendiff`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff` ou `gvimdiff`. Si KDiff3 ne vous intéresse pas pour gérer les différences mais seulement pour la résolution de fusion et qu'il est présent dans votre chemin d'exécution, vous pouvez lancer

```
$ git config --global merge.tool kdiff3
```

Si vous lancez ceci au lieu de modifier les fichiers `extMerge` ou `extDiff`, Git utilisera KDif3 pour les résolutions de fusion et l'outil diff normal de Git pour les différences.

7.1.4 Formatage and espaces blancs

Les problèmes de formatage et de blancs font partie des plus subtiles et frustrants que les développeurs rencontrent lorsqu'ils collaborent, spécifiquement sur plusieurs plates-formes. Il est très facile d'introduire des modifications subtiles de blancs lors de soumission de patchs ou d'autres modes de collaboration car les éditeurs de textes les insèrent silencieusement ou les programmeurs Windows ajoutent de retour chariot à la fin des lignes qu'il modifient. Git dispose de quelques options de configuration pour traiter ces problèmes.

core.autocrlf

Si vous programmez vous-même sous Windows ou si vous utilisez un autre système d'exploitation mais devez travailler avec des personnes travaillant sous Windows, vous rencontrerez à un moment ou à un autre des problèmes de caractères de fin de ligne. Ceci est dû au fait que Windows utilise pour marquer les fins de ligne dans ses fichiers un caractère « retour chariot » (*carriage return*, CR) suivi d'un caractère « saut de ligne » (*line feed*, LF), tandis que Mac et Linux utilisent seulement le caractère « saut de ligne ». C'est un cas subtile mais incroyablement ennuyeux de problème généré par la collaboration inter plate-forme.

Git peut gérer ce cas en convertissant automatiquement les fins de ligne CRLF en LF lorsque vous validez, et inversement lorsqu'il extrait des fichiers sur votre système. Vous pouvez activer cette fonctionnalité au moyen du paramètre `core.autcrlf`. Si vous avez une machine Windows, positionnez-le à `true`. Git convertira les fins de ligne de LF en CRLF lorsque vous extrayerez votre code :

```
$ git config --global core.autocrlf true
```

Si vous utilisez un système Linux ou Mac qui utilise les fins de ligne LF, vous ne souhaitez sûrement pas que Git les convertisse automatiquement lorsque vous extrayez des fichiers. Cependant, si un fichier contenant des CRLF est accidentellement introduit en version, vous souhaitez que Git le corrige . Vous pouvez indiquer à Git de convertir CRLF en LF lors de la validation mais pas dans l'autre sens en fixant `core.autocrlf` à `input` :

```
$ git config --global core.autocrlf input
```

Ce réglage devrait donner des fins de ligne en CRLF lors d'extraction sous Windows mais en LF sous Mac et Linux et dans le dépôt.

Si vous êtes un programmeur Windows gérant un projet spécifique à Windows, vous pouvez désactiver cette fonctionnalité et forcer l'enregistrement des « retour chariot » dans le dépôt en réglant la valeur du paramètre à `false` :

```
$ git config --global core.autocrlf false
```

core.whitespace

Git est paramétré par défaut pour détecter et corriger certains problèmes de blancs. Il peut rechercher quatre problèmes de base de blancs. La correction de deux problèmes est activée par défaut et peut être désactivée et celle des deux autres n'est pas activée par défaut mais peut être activée.

Les deux activées par défaut sont `trailing-space` qui détecter les espaces en fin de ligne et `space-before-tab` qui recherche les espaces avant les tabulations au début d'une ligne.

Les deux autres qui sont désactivées par défaut mais peuvent être activées sont `indent-with-non-tab` qui recherche des lignes qui commencent par huit espaces ou plus au lieu de tabulations et `cr-at-eol` qui indique à Git que les « retour chariot » en fin de ligne sont acceptés.

Vous pouvez indiquer à Git quelle correction vous voulez activer en fixant `core.whitespace` avec les valeurs que vous voulez ou non, séparées par des virgules. Vous pouvez désactiver des réglages en les éliminant de la chaîne de paramétrage ou en les préfixant avec un `-`. Par exemple, si vous souhaiter activer tout sauf `cr-at-eol`, vous pouvez lancer ceci :

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git va détecter ces problèmes quand vous lancez une commande `git diff` et essayer de les coloriser pour vous permettre de les régler avant de valider. Il utilisera aussi ces paramètres pour vous aider quand vous appliquerez des patchs avec `git apply`. Quand vous appliquez des patchs, vous pouvez paramétrer Git pour qu'il vous avertisse s'il doit appliquer des patchs qui présentent les défauts de blancs :

```
$ git apply --whitespace=warn <patch>
```

Ou vous pouvez indiquer à Git d'essayer de corriger automatiquement le problème avant d'appliquer le patch :

```
$ git apply --whitespace=fix <patch>
```

Ces options s'appliquent aussi à `git rebase`. Si vous avez validé avec des problèmes de blancs mais n'avez pas encore poussé en amont, vous pouvez lancer un `rebase` avec l'option `--whitespace=fix` pour faire corriger à Git les erreurs de blancs pendant qu'il réécrit les patchs.

7.1.5 Configuration du serveur

Il n'y a pas autant d'options de configuration de Git côté serveur, mais en voici quelques unes intéressantes dont il est utile de prendre note.

receive.fsckObjects

Par défaut, Git ne vérifie pas la cohérence entre les objets qu'on lui pousse. Bien que Git puisse vérifier que chaque objet correspond bien à sa somme de contrôle et pointe vers des objets valides, il ne le fait pas par défaut sur chaque poussée. C'est une opération relativement lourde qui peut énormément allonger les poussées selon la taille du dépôt ou de la poussée. Si vous voulez que Git vérifie la cohérence des objets à chaque poussée, vous pouvez le forcer en fixant le paramètre `receive.fsckObjects` à true :

```
$ git config --system receive.fsckObjects true
```

Maintenant, Git va vérifier l'intégrité de votre dépôt avant que chaque poussée ne soit acceptée pour s'assurer que des clients défectueux n'introduisent pas des données corrompues.

receive.denyNonFastForwards

Si vous rebasez des commits que vous avez déjà poussés, puis essayez de pousser à nouveau, ou inversemement, si vous essayez de pousser un commit sur une branche distante qui ne contient pas le commit sur lequel la branche distante pointe, votre essai échouera. C'est généralement une bonne politique, mais dans le cas d'un rebasage, vous pouvez décider que vous savez ce que vous faîtes et forcer la mise à jour de la branche distante en ajoutant l'option `-f` à votre commande.

Pour désactiver la possibilité de forcer la mise à jour des branches distantes vers des références pas en avance rapide, réglez `receive.denyNonFastForwards` :

```
$ git config --system receive.denyNonFastForwards true
```

L'autre moyen d'obtenir ce résultat réside dans les crochets de réception côté-serveur, qui seront abordés en seconde partie. Cette approche vous permet de faire des choses plus complexes tel qu'interdire les modifications sans avance rapide à un certain groupe d'utilisateurs.

receive.denyDeletes

Un contournement possible de la politique `denyNonFastForwards` consiste à effacer la branche puis à la repousser avec ses nouvelles références. Dans les versions les plus récentes de Git (à partir de la version 1.6.1), vous pouvez régler `receive.denyDeletes` à true :

```
$ git config --system receive.denyDeletes true
```

Cela interdit totalement l'effacement de branche et de balise. Aucun utilisateur n'en a le droit. Pour pouvoir effacer des branches distantes, vous devez effacer manuellement les fichiers de référence

sur le serveur. Il existe aussi des moyens plus intéressants de gérer cette politique utilisateur par utilisateur au moyen des listes de contrôle d'accès, point qui sera abordé à la fin de ce chapitre.

7.2 Attributs Git

Certains de ces réglages peuvent aussi s'appliquer sur un chemin, de telle sorte que Git ne les applique que sur un sous-répertoire ou un sous-ensemble de fichiers. Ces réglages par chemin sont appelés attributs Git et sont définis soit dans une fichier `.gitattributes` dans un répertoire (normalement la racine du projet), soit dans un fichier `.git/info/attributes` si vous ne souhaitez pas que la fichier de description des attributs fasse partie du projet.

Les attributs permettent de spécifier des stratégies de fusion différentes pour certains fichiers ou répertoires dans votre projet, d'indiquer à Git la manière de calculer les différences pour certains fichiers non-texte, ou de faire filtrer à Git le contenu avant qu'il ne soit validé ou extrait. Dans ce chapitre, nous traiterons certains attributs applicables aux chemins et détaillerons quelques exemples de leur utilisation en pratique.

7.2.1 Fichiers binaires

Un des trucs malins auxquels les attributs Git sont utilisés est d'indiquer à Git quels fichiers sont binaires (dans les cas où il ne pourrait pas le deviner par lui-même) et de lui donner les instructions spécifiques pour les traiter. Par exemple, certains fichiers peuvent être générés par machine et impossible à traiter par diff, tandis que pour certains autres fichiers binaires, les différences peuvent être calculées. Nous détaillerons comment indiquer à Git l'un et l'autre.

Identification des fichiers binaires

Certains fichiers ressemblent à des fichiers texte mais doivent en tout état de cause être traités comme des fichiers binaires. Par exemple, les projets Xcode sous Mac contiennent un fichier finissant en `.pbxproj`, qui est en fait un jeu de données JSON (format de données en texte javascript) enregistré par l'application EDI pour y sauver les réglages entre autres de compilation. Bien que ce soit techniquement un fichier texte en ASCII, il n'y a aucun intérêt à le gérer comme tel parce que c'est en fait une mini base de données. Il est impossible de fusionner les contenus si deux utilisateurs le modifient et les calculs de différence par défaut sont inutiles. Ce fichier n'est destiné qu'à être manipulé par un programme En résumé, ce fichier doit être considéré comme un fichier binaire.

Pour indiquer à Git de traiter tous les fichiers `pbxproj` comme binaires, ajoutez la ligne suivante à votre fichier `.gitattributes` :

```
*.pbxproj -crlf -diff
```

À présent, Git n'essaiera pas de convertir ou de corriger les problèmes des CRLF, ni de calculer ou d'afficher les différences pour ces fichiers quand vous lancez git show ou git diff sur votre projet. Dans la branche 1.6 de Git, vous pouvez aussi utiliser une macro fournie qui signifie `-crlf -diff` :

```
*.pbxproj binary
```

Comparaison de fichiers binaires

Dans la branche 1.6 de Git, vous pouvez utiliser la fonctionnalité des attributs Git pour effectivement comparer les fichiers binaires. Pour ce faire, indiquez à Git comment convertir vos données binaires en format texte qui peut être comparé via un diff normal.

Comme c'est une fonctionnalité plutôt cool et peu connue, nous allons en voir quelques exemples. Premièrement, nous utiliserons cette technique pour résoudre un des problèmes les plus ennuyeux de l'humanité : gérer en contrôle de version les document Word. Tout le monde convient que Word est l'éditeur de texte le plus horrible qui existe, mais bizarrement, tout le monde persiste à l'utiliser. Si vous voulez gérer en version des documents Word, vous pouvez les coller dans un dépôt Git et les valider de temps à autre. Mais qu'est-ce que ça vous apporte ? Si vous lancez `git diff` normalement, vous verrez quelque chose comme :

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

Vous ne pouvez pas comparer directement les versions à moins de les extraire et de les parcourir manuellement. En fait, vous pouvez faire la même chose plutôt bien en utilisant les attributs Git. Ajoutez la ligne suivante dans votre fichier `.gitattributes` :

```
*.doc diff=word
```

Cette ligne indique à Git que tout fichier correspondant au patron (.doc) doit utiliser le filtre `word` pour visualiser le diff des modifications. Qu'est-ce que le filtre « word » ? Nous devons le définir. Vous allez configurer Git à utiliser le programme `strings` pour convertir les documents Word en fichiers texte lisibles qu'il pourra alors comparer correctement :

```
$ git config diff.word.textconv strings
```

À présent, Git sait que s'il essaie de faire un diff entre deux instantanés et qu'un des fichiers finit en `.doc`, il devrait faire passer ces fichiers par le filtre `word` définit comme le programme `strings`. Cette méthode fait effectivement des jolies versions texte de vos fichiers Word avant d'essayer de les comparer.

Voici un exemple. J'ai mis le chapitre 1 de ce livre dans Git, ajouté du texte à un paragraphe et sauvegardé le document. Puis, j'ai lancé `git diff` pour visualiser ce qui a changé :

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
```

```
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
 re going to cover how to get it and set it up for the first time if you don
 t already have it on your system.
 In Chapter Two we will go over basic Git usage - how to use Git for the 80%
-s going on, modify stuff and contribute changes. If the book spontaneously
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let's see if this works.
```

Git réussit à m'indiquer succinctement que j'ai ajouté la chaîne « *Let's see if this works* », ce qui est correct. Ce n'est pas parfait, car il y a toujours un tas de données aléatoires à la fin, mais c'est suffisant. Si vous êtes capable d'écrire un convertisseur Word vers texte qui fonctionne suffisamment bien, cette solution peut s'avérer très efficace. Cependant, `strings` est disponible sur la plupart des systèmes Mac et Linux et peut donc constituer un bon début pour de nombreux formats binaires.

Un autre problème intéressant concerne la comparaison de fichiers d'images. Une méthode consiste à faire passer les fichiers JPEG à travers un filtre qui extrait les données EXIF, les méta-données enregistrées avec la plupart de formats d'image. Si vous téléchargez et installez le programme `exiftool`, vous pouvez l'utiliser pour convertir vos images en texte de méta-données de manière que le diff puisse au moins montrer une représentation textuelle des modifications pratiquées :

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Si vous remplacez une image dans votre projet et lancez `git diff`, vous verrez ceci :

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

Vous pouvez réaliser rapidement que la taille du fichier et les dimensions des images ont toutes deux changé.

7.2.2 Expansion des mots-clés

L'expansion de mots-clés dans le style de CVS ou de SVN est souvent une fonctionnalité demandée par les développeurs qui y sont habitués. Le problème principal de ce système avec Git et que vous ne pouvez pas modifier un fichier avec l'information concernant le commit après la validation parce que Git calcule justement la somme de contrôle sur son contenu. Cependant, vous pouvez injecter des informations textuelles dans un fichier au moment où il est extrait et les retirer avant qu'il ne soit ajouté à une validation. Les attributs Git vous fournissent deux manières de le faire.

Premièrement, vous pouvez injecter automatiquement la somme de contrôle SHA-1 d'un blob dans un champ `$Id$` d'un fichier. Si vous positionnez cet attribut pour un fichier ou un ensemble de fichiers, la prochaine fois que vous extrairez cette branche, Git remplacera chaque champ avec le SHA-1 du blob. Il est à noter que ce n'est pas le SHA du commit mais celui du blob lui-même :

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

À la prochaine extraction de ce fichier, Git injecte le SHA du blob :

```
$ rm text.txt
$ git checkout -- text.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Néanmoins, ce résultat n'a que peu d'intérêt. Si vous avez utilisé la substitution avec CVS ou Subversion, il est possible d'inclure la date. Le code SHA n'est pas des plus utiles car il est plutôt aléatoire et ne vous permet pas de distinguer si tel SHA est plus récent ou ancien que tel autre.

Il apparaît que vous pouvez écrire vos propres filtres pour réaliser des substitutions dans les fichiers lors des validations/extractions. Ces filtres s'appellent « *clean* » et « *smudge* ». Dans le fichier `.gitattributes`, vous pouvez indiquer un filtre pour des chemins particuliers puis créer des scripts qui traiterons ces fichiers avant qu'ils soient validés (« *clean* », voir figure 7-2) et juste avant qu'il soient extraits (« *smudge* », voir figure 7-3). Ces filtres peuvent servir à faire toutes sortes de choses attrayantes.

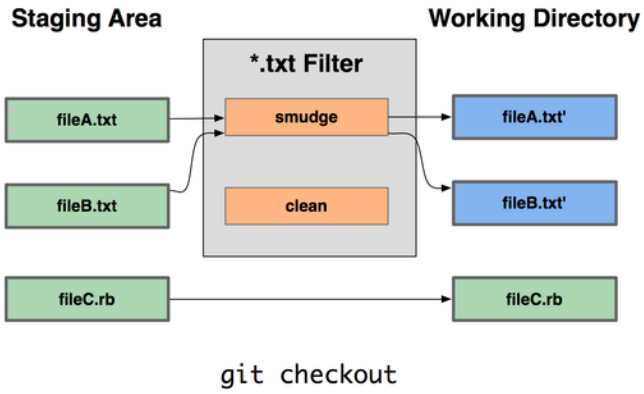

Figure 7.2: *Le filtre « smudge » est lancé lors d'une extraction.*

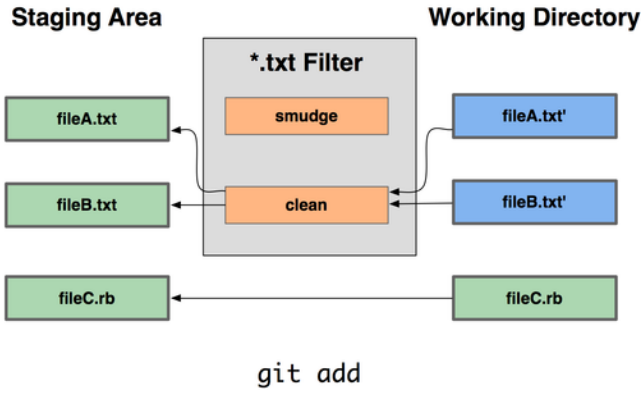

Figure 7.3: *Le filtre « clean » est lancé lorsque les fichiers sont indexés.*

Le message de validation d'origine pour cette fonctionnalité donne un exemple simple permettant de passer tout votre code C par le programme `indent` avant de valider. Vous pouvez le faire en réglant l'attribut `filter` dans votre fichier `.gitattributes` pour filtrer les fichiers `*.c` avec le filtre « indent » :

```
*.c     filter=indent
```

Ensuite, indiquez à Git ce que le filtre « indent » fait sur smudge et clean :

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Dans ce cas, quand vous validez des fichiers qui correspondent à `*.c`, Git les fera passer par le programme `indent` avant de les valider et les fera passer par le programme `cat` avant de les extraire sur votre disque. Le programme `cat` ne fait rien : il se contente de régurgiter les données telles qu'il les a lues. Cette combinaison filtre effectivement tous les fichiers de code source C par `indent` avant leur validation.

Un autre exemple intéressant fournit l'expansion du mot-clé `$Date$` dans le style RCS. Pour le réaliser correctement, vous avez besoin d'un petit script qui prend un nom de fichier, calcule la date de la dernière validation pour le projet et l'insère dans le fichier. Voici un petit script Ruby qui le fait :

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Tout ce que le script fait, c'est récupérer la date de la dernière validation à partir de la commande `git log`, la coller dans toutes les chaînes `$Date$` qu'il trouve et afficher le résultat. Ce devrait être simple dans n'importe quel langage avec lequel vous êtes à l'aise. Si vous appelez ce fichier `expand_date` et que vous le placez dans votre chemin. À présent, il faut paramétrer un filtre dans

Git (appelons le `dater`) et lui indiquer d'utiliser le filtre `expand_date` en tant que `smudge` sur les fichiers à extraire. Nous utiliserons une expression Perl pour nettoyer lors d'une validation :

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Cette commande Perl extrait tout ce qu'elle trouve dans une chaîne `$Date$` et la réinitialise. Le filtre prêt, on peut le tester en écrivant le mot-clé `$Date$` dans un fichier, puis en créant un attribut Git pour ce fichier qui fait référence au nouveau filtre :

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Si vous validez ces modifications et extrayez le fichier à nouveau, vous remarquez le mot-clé correctement substitué :

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Vous pouvez voir à quel point cette technique peut être puissante pour des applications personnalisées. Il faut rester néanmoins vigilant car le fichier `.gitattributes` est validé et inclus dans le projet tandis que le gestionnaire (ici, `dater`) ne l'est pas. Du coup, ça ne marchera pas partout. Lorsque vous créez ces filtres, ils devraient pouvoir avoir un mode dégradé qui n'empêche pas le projet de fonctionner.

7.2.3 Export d'un dépôt

Les données d'attribut Git permettent aussi de faire des choses intéressantes quand vous exportez une archive du projet.

export-ignore

Vous pouvez dire à Git de ne pas exporter certains fichiers ou répertoires lors de la génération d'archive. S'il y a un sous-répertoire ou un fichier que vous ne souhaitez pas inclure dans le fichier archive mais que vous souhaitez extraire dans votre projet, vous pouvez indiquer ce fichier via l'attribut `export-ignore`.

Par exemple, disons que vous avez des fichiers de test dans le sous-répertoire `test/` et que ce n'est pas raisonnable de les inclure dans l'archive d'export de votre projet. Vous pouvez ajouter la ligne suivante dans votre fichier d'attribut Git :

```
test/ export-ignore
```

À présent, quand vous lancez git archive pour créer une archive `tar` de votre projet, ce répertoire ne sera plus inclus dans l'archive.

export-subst

Une autre chose à faire pour vos archives est une simple substitution de mots-clés. Git vous permet de placer la chaîne `$Format:$` dans n'importe quel fichier avec n'importe quel code de format du type `--pretty=format` que vous avez pu voir au chapitre 2. Par exemple, si vous voulez inclure un fichier appelé `LAST_COMMIT` dans votre projet et y injecter automatiquement la date de dernière validation lorsque `git archive` est lancé, vous pouvez créer un fichier comme ceci :

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Quand vous lancez `git archive`, le contenu de ce fichier inclus dans l'archive ressemblera à ceci :

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

7.2.4 Stratégies de fusion

Vous pouvez aussi utiliser les attributs Git pour indiquer à Git d'utiliser des stratégies de fusion différenciées pour des fichiers spécifiques dans votre projet. Une option très utile est d'indiquer à Git de ne pas essayer de fusionner des fichiers spécifiques quand ils rencontrent des conflits mais plutôt d'utiliser prioritairement votre version du fichier.

C'est très utile si une branche de votre projet a divergé ou s'est spécialisée, mais que vous souhaitez pouvoir fusionner les modifications qu'elle porte et vous voulez ignorer certains fichiers. Supposons que vous avez un fichier de paramètres de base de données appelé database.xml différent sur deux branches et vous voulez les fusionner sans corrompre le fichier de base de données. Vous pouvez déclarer un attribut comme ceci :

```
database.xml merge=ours
```

Si vous fusionnez dans une autre branche, plutôt que de rencontrer des conflits de fusion avec le fichier database.xml, vous verrez quelque chose comme :

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Dans ce cas, database.xml reste dans l'état d'origine, quel qu'il soit.

7.3 Crochets Git

Comme de nombreux autres systèmes de gestion de version, Git dispose d'un moyen de lancer des scripts personnalisés quand certaines actions importantes ont lieu. Il y a deux groupes de crochets : ceux côté client et ceux côté serveur. Les crochets côté client concernent les opérations de client telles que la validation et la fusion. Les crochets côté serveur concernent les opérations de serveur Git telles que la réception de commits. Vous pouvez utiliser ces crochets pour toutes sortes de raisons dont nous allons détailler quelques unes.

7.3.1 Installation d'un crochet

Les crochets sont tous stockés dans le sous-répertoire `hooks` du répertoire Git. Dans la plupart des projets, c'est `.git/hooks`. Par défaut, Git popule ce répertoire avec quelques scripts d'exemple déjà utiles par eux-mêmes ; mais ils servent aussi de documentation sur les paramètres de chaque script. Tous les exemples sont des scripts shell avec un peu de Perl mais n'importe quel script exécutable nommé correctement fonctionnera. Vous pouvez les écrire en Ruby ou Python ou ce que vous voudrez. Pour les versions de Git postérieures à 1.6, ces fichiers crochet d'exemple se terminent en `.sample` et il faudra les renommer. Pour les versions de Git antérieures à 1.6, les fichiers d'exemple sont nommés correctement mais ne sont pas exécutables.

Pour activer un script de crochet, placez un fichier dans le sous-répertoire `hook` de votre répertoire Git, nommé correctement et exécutable. À partir de ce moment, il devrait être appelé. Abordons donc les noms de fichiers hooks les plus importants.

7.3.2 Crochets côté client

Il y a de nombreux crochets côté client. Ce chapitre les classe entre crochets de traitement de validation, scripts de traitement par e-mail et le reste des scripts côté client.

Crochets de traitement de validation

Les quatre premiers crochets ont trait au processus de validation. Le crochet `pre-commit` est lancé en premier, avant même que vous ne saisissiez le message de validation. Il est utilisé pour inspecter l'instantané qui est sur le point d'être validé, pour vérifier si vous avez oublié quelque chose, pour s'assurer que les tests passent ou pour examiner ce que vous souhaitez inspecter dans le code. Un code de sortie non nul de ce crochet annule la validation, bien que vous puissiez le contourner avec `git commit --no-verify`. Vous pouvez réaliser des actions telles qu'une vérification de style (en utilisant lint ou un équivalent), d'absence de blancs en fin de ligne (le crochet par défaut fait exactement cela) ou de documentation des nouvelles méthodes.

Le crochet `prepare-commit-msg` est appelé avant que l'éditeur de message de validation ne soit lancé après que le message par défaut a été créé. Il vous permet d'éditer le message par

défaut avant que l'auteur ne le voit. Ce crochet accepte quelques options : le chemin du fichier qui contient le message de validation actuel, le type de validation et le SHA-1 du commit si c'est un commit amendé. Ce crochet ne sert généralement à rien pour les validations normales. Par contre, il est utile pour les validations où le message par défaut est généré, tel que les modèles de message de validation, les validations de fusion, les commits écrasés ou amendés. Vous pouvez l'utiliser en conjonction avec un modèle de messages pour insérer de l'information par programme.

Le crochet `commit-msg` accepte un paramètre qui est encore le chemin du fichier temporaire qui contient le message de validation actuel. Si ce script rend un code de sortie non nul, Git abandonne le processus de validation, ce qui vous permet de vérifier l'état de votre projet ou du message de validation avant de laisser passer un commit. Dans la dernière section de ce chapitre, l'utilisation de ce crochet permettra de vérifier que le message de validation est conforme à un format obligatoire.

Après l'exécution du processus complet de validation, le crochet `post-commit` est appelé. Il n'accepte aucun argument mais vous pouvez facilement accéder au dernier commit grâce à `git log -1 HEAD`. Généralement, ce script sert à réaliser des notifications ou des choses similaires.

Les scripts de gestion de validation côté client peuvent être utilisés pour n'importe quelle méthode de travail. Ils sont souvent utilisés pour mettre en œuvre certaines politiques, bien qu'il faille noter que ces scripts ne sont pas transférés lors d'un clonage. Vous pouvez faire appliquer les politiques de gestion au niveau serveur pour rejeter les poussées de commits qui ne sont pas conformes à certaines règles, mais il reste complètement du ressort du développeur de les utiliser côté client. Ce sont des scripts destinés à aider les développeurs et ils doivent être mis en place et maintenus par ces derniers qui peuvent tout aussi bien les outrepasser ou les modifier à tout moment.

Crochets de gestion e-mail

Vous pouvez régler trois crochets côté client pour la gestion à base d'e-mail. Ils sont tous invoqués par la commande `git am`, donc si vous n'êtes pas habitués à utiliser cette commande dans votre mode de gestion, vous pouvez simplement passer la prochaine section. Si vous acceptez des patchs préparés par `git format-patch` par e-mail, alors certains de ces crochets peuvent vous être très utiles.

Le premier crochet lancé est `applypatch-msg`. Il accepte un seul argument : le nom du fichier temporaire qui contient le message de validation proposé. Git abandonne le patch si ce script sort avec un code non nul. Vous pouvez l'utiliser pour vérifier que la message de validation est correctement formaté ou pour normaliser le message en l'éditant sur place par script.

Le crochet lancé ensuite lors de l'application de patchs via `git am` s'appelle `pre-applypatch`. Il n'accepte aucun argument et est lancé après que le patch a été appliqué, ce qui vous permet d'inspecter l'instantané avant de réaliser la validation. Vous pouvez lancer des tests ou inspecter l'arborescence active avec ce script. S'il manque quelque chose ou que les tests ne passent pas, un code de sortie non nul annule la commande `git am` sans valider le patch.

Le dernier crochet lancé pendant l'opération `git am` s'appelle `post-applypatch`. Vous pouvez l'utiliser pour notifier un groupe ou l'auteur du patch que vous venez de l'appliquer. Vous ne pouvez plus arrêter le processus de validation avec ce script.

Autres crochets côté client

Le crochet `pre-rebase` est invoqueé avant que vous ne rebasiez et peut interrompre le processus s'il sort avec un code d'erreur non nul. Vous pouvez utiliser ce crochet pour empêcher de

rebase tout commit qui a déjà été poussé. C'est ce que fait le crochet d'exemple `pre-rebase` que Git installe, même s'il considère que la branche cible de publication s'appelle `next`. Il est très probable que vous ayez à changer ce nom pour celui que vous utilisez réellement en branche publique stable.

Après avoir effectué avec succès un `git checkout`, la crochet `post-chechout` est lancé. Vous pouvez l'utiliser pour paramétrer correctement votre environnement projet dans votre copie de travail. Cela peut signifier y déplacer des gros fichiers binaires que vous ne souhaitez pas voir en gestion de source, générer automatiquement la documentation ou quelque chose dans le genre.

Enfin, le crochet `post-merge` s'exécute à la suite d'une commande `merge` réussie. Vous pouvez l'utiliser pour restaurer certaines données non gérées par Git dans le copie de travail telles que les informations de permission. Ce crochet permet de même de valider la présence de fichiers externes au contrôle de Git que vous souhaitez voir recopiés lorsque la copie de travail change.

7.3.3 Crochets côté serveur

En complément des crochets côté client, vous pouvez utiliser comme administrateur système quelques crochets côté serveur pour appliquer quasiment toutes les règles de votre projet. Ces scripts s'exécutent avant et après chaque poussées sur le serveur. Les crochets `pre` peuvent rendre un code d'erreur non nul à tout moment pour rejeter la poussée et afficher un message d'erreur au client. Vous pouvez mettre en place des règles aussi complexes que nécessaire.

pre-receive et post-receive

Le premier script lancé lors de la gestion d'une poussée depuis un client est `pre-receive`. Il accepte une liste de références lues sur stdin. S'il sort avec un code d'erreur non nul, aucune n'est acceptée. Vous pouvez utiliser ce crochet pour réaliser des tests tels que s'assurer que toutes les références mises à jour le sont en avance rapide ou pour s'assurer que l'utilisateur dispose bien des droits de création, poussée, destruction ou de lecture des mises à jour pour tous les fichiers qu'il cherche à mettre à jour dans cette poussée.

Le crochet `post-receive` est lancé après l'exécution complète du processus et peut être utilisé pour mettre à jour d'autres services ou pour notifier des utilisateurs. Il accepte les même données sur stdin que `pre-receive`. Il peut par exemple envoyer un e-mail à une liste de diffusion, notifier un serveur d'intégration continue ou mettre à jour un système de suivi de tickets. Il peut aussi analyser les messages de validation à la recherche d'ordres de mise à jour de l'état des tickets. Ce script ne peut pas arrêter le processus de poussée mais le client n'est pas déconnecté tant qu'il n'a pas terminé. Il faut donc être prudent à ne pas essayer de lui faire réaliser des actions qui peuvent durer longtemps.

update

Le script `update` est très similaire au script `pre-receive`, à la différence qu'il est lancé une fois par branche qui doit être modifiée lors de la poussée. Si la poussée s'applique à plusieurs branches, `pre-receive` n'est lancé qu'une fois, tandis qu'`update` est lancé une fois par branche impactée. Au lieu de lire à partir de stdin, ce script accepte trois arguments : le nom de la référence (branche), le SHA-1 du commit pointé par la référence avant la poussée et le SHA-1 que l'utilisateur est en train de pousser. Si le script `update` se termine avec un code d'erreur non nul, seule la référence est rejetée. Les autres références pourront être mises à jour.

7.4 Exemple de politique gérée par Git

Dans ce chapitre, nous allons utiliser ce que nous venons d'apprendre pour installer une gestion Git qui vérifie la présence d'un format personnalisé de message de validation, n'autorise que les poussées en avance rapide et autorise seulement certains utilisateurs à modifier certains sous-répertoires dans un projet. Nous construirons des scripts client pour informer les développeurs que leurs poussées vont être rejetées et des scripts sur le serveur pour mettre effectivement en place ces règles.

J'ai utilisé Ruby pour les écrire, d'abord parce que c'est mon langage de script favori, ensuite parce que je pense que c'est le langage de script qui s'apparente le plus à du pseudo-code. Ainsi, il devrait être simple de suivre grossièrement le code même sans connaître le langage Ruby. Cependant, tout langage peut être utilisé. Tous les scripts d'exemple distribués avec Git sont soit en Perl soit en Bash, ce qui donne de nombreux autres exemples de crochets dans ces langages.

7.4.1 Crochets côté serveur

Toutes les actions côté serveur seront contenues dans le fichier `update` dans le répertoire `hooks`. Le fichier `update` s'exécute une fois par branche poussée et accepte comme paramètre la référence sur laquelle on pousse, l'ancienne révision de la branche et la nouvelle révision de la branche. Vous pouvez aussi avoir accès à l'utilisateur qui pousse si la poussée est réalisée par SSH. Si vous avez permis à tout le monde de se connecter avec un utilisateur unique (comme « git ») avec une authentification à clef publique, il vous faudra fournir à cet utilisateur une enveloppe de shell qui déterminera l'identité de l'utilisateur à partir de sa clef publique et positionnera une variable d'environnement spécifiant cette identité. Ici, je considère que la variable d'environnement `$USER` indique l'utilisateur connecté, donc le script update commence par rassembler toutes les informations nécessaires :

```
#!/usr/bin/env ruby

$nomref       = ARGV[0]
$anciennerev  = ARGV[1]
$nouvellerev  = ARGV[2]
$utilisateur  = ENV['USER']

puts    "Vérification    des    règles...    \n(#{$nomref})    (#{$anciennerev[0,
6]}) (#{$nouvellerev[0,6]})"
```

Et oui, j'utilise des variables globales. C'est seulement pour simplifier la démonstration.

Application d'une politique de format du message de validation

Notre première tâche consiste à forcer que chaque message de validation adhère à un format particulier. En guise d'objectif, obligeons chaque message à contenir une chaîne de caractère qui ressemble à « ref: 1234 » parce que nous souhaitons que chaque validation soit liée à une tâche de notre système de tickets. Nous devons donc inspecter chaque commit poussé, vérifier la présence de la chaîne et sortir avec un code non-nul en cas d'absence pour rejeter la poussée.

Vous pouvez obtenir une liste des valeurs SHA-1 de tous les commits en cours de poussée en passant les valeurs `$nouvellerev` et `$anciennerev` à une commande de plomberie Git

appelée `git-rev-list`. C'est comme la commande `git log` mais elle n'affiche par défaut que les valeurs SHA-1, sans autre information. Donc, pour obtenir une liste de tous les SHA des commits introduits entre un SHA de commit et un autre, il suffit de lancer quelque chose comme :

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Vous pouvez récupérer la sortie, boucler sur chacun de ces SHA de commit, en extraire le message et tester la conformance du message avec une structure au moyen d'une expression rationnelle.

Vous devez trouver comment extraire le message de validation à partir de chacun des commits à tester. Pour accéder aux données brutes du commit, vous pouvez utiliser une autre commande de plomberie appelée `git cat-file`. Nous traiterons en détail toutes ces commandes de plomberie au chapitre 9 mais pour l'instant, voici ce que cette commande affiche:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Un moyen simple d'extraire le message de validation d'un commit à partir de son SHA-1 consiste à rechercher la première ligne vide et à sélectionner tout ce qui suit. Cela peut être facilement réalisé avec la commande `sed` sur les systèmes Unix :

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Vous pouvez utiliser cette ligne pour récupérer le message de validation de chaque commit en cours de poussée et sortir si quelque chose ne correspond à ce qui est attendu. Pour sortir du script et rejeter la poussée, il faut sortir avec un code non nul. La fonction complète ressemble à ceci :

```
$regex = /\[ref: (\d+)\]/

# vérification du format des messages de validation
def verif_format_message
  revs_manquees = `git rev-list #{$anciennerev}..#{$nouvellerev}`.split("\n")
  revs_manquees.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
```

```
      puts "[REGLE] Le message de validation ne suit pas le format"
      exit 1
    end
  end
end
verif_format_message
```

Placer ceci dans un script `update` rejettera les mises à jour contenant des commits dont les messages ne suivent pas la règle.

Mise en place d'un système d'ACL par utilisateur

Supposons que vous souhaitiez ajouter un mécanisme à base de liste de contrôle d'accès (access control list : ACL) qui permette de spécifier quel utilisateur a le droit de pousser des modifications vers quelle partie du projet. Certains personnes ont un accès complet tandis que d'autres n'ont accès que pour mettre à jour certains sous-répertoires ou certains fichiers. Pour faire appliquer ceci, nous allons écrire ces règles dans un fichier appelé `acl` situé dans le dépôt brut Git sur le serveur. Le crochet `update` examinera ces règles, listera les fichiers impactés par la poussée et déterminera si l'utilisateur qui pousse a effectivement les droits nécessaires sur ces fichiers.

Écrivons en premier le fichier d'ACL. Nous allons utiliser un format très proche de celui des ACL de CVS. Le fichier est composé de lignes dont le premier champ est `avail` ou `unavail`, le second est une liste des utilisateurs concernés séparés par des virgules t le dernier champ indique le chemin pour lequel la règle s'applique (le champ vide indiquant une règle générale). Tous les champs sont délimités par un caractère pipe « | ».

Dans notre cas, il y a quelques administrateurs, des auteurs de documentation avec un accès au répertoire `doc` et un développeur qui n'a accès qu'aux répertoires `lib` et `tests`. Le fichier ACL ressemble donc à ceci :

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Le traitement consiste à lire le fichier dans une structure utilisable. Dans notre cas, pour simplifier, nous ne traiterons que les directives `avail`. Voici une fonction qui crée à partir du fichier un tableau associatif dont la clé est l'utilisateur et la valeur une liste des chemins pour lesquels l'utilisateur a les droits en écriture :

```
def get_acl_access_data(nom_fichier_acl)
  # lire le fichier ACL
  fichier_acl = File.read(nom_fichier_acl).split("\n").reject { |ligne| ligne == '' }
  acces = {}
  fichier_acl.each do |ligne|
    avail, utilisateurs, chemin = ligne.split('|')
    next unless avail == 'avail'
    utilisateurs.split(',').each do |utilisateur|
```

```
      acces[utilisateur] ||= []
      acces[utilisateur] << chemin
    end
  end
  acces
end
```

Pour le fichier d'ACL décrit plus haut, le fonction `get_acl_access_data` retourne une structure de données qui ressemble à ceci :

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

En plus des permissions, il faut déterminer les chemins impactés par la poussée pour s'assurer que l'utilisateur a bien droit d'y toucher.

La liste des fichiers modifiés est assez simplement obtenue par la commande `git log` complétée par l'option `--name-only` mentionnée au chapitre 2.

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Chaque fichier des commits doit être vérifié par rapport à la structure ACL retournée par la fonction `get_acl_access_data` pour déterminer si l'utilisateur a le droit de pousser tous ses commits :

```
# permission à certains utilisateurs de modifier certains sous-répertoires du projet
def verif_perms_repertoire
  acces = get_acl_access_data('acl')

  # verifier si quelqu'un chercher à pousser où il n'a pas le droit
  nouveaux_commits = `git rev-list #{$anciennerev}..#{$nouvellerev}`.split("\n")
  nouveaux_commits.each do |rev|
   fichiers_modifies = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    fichiers_modifies.each do |chemin|
      next if chemin.size == 0
      acces_permis = false
      acces[$utilisateur].each do |chemin_acces|
        if !chemin_acces  # l'utilisateur a un accès complet
          || (chemin.index(chemin_acces) == 0) # acces à ce chemin
```

```
          acces_permis = true
        end
      end
      if !acces_permis
        puts "[ACL] Vous n'avez pas le droit de pousser sur #{path}"
        exit 1
      end
    end
  end
end

verif_perms_repertoire
```

L'algorithme ci-dessus reste simple. Pour chaque élément de la liste des nouveaux commits à pousser obtenue au moyen de `git rev-list`, on vérifie que l'utilisateur qui pousse a accès au chemin de chacun des fichiers modifiés. L'expression `chemin.index(chemin_acces) == 0` est un Rubyisme qui n'est vrai que si `chemin` commence comme `chemin_acces`. Ce script s'assure non pas qu'un `chemin` fait partie des chemins permis, mais que tous les chemins accédés font bien partie des chemins permis.

À présent, les utilisateurs ne peuvent plus pousser de commits comprenant un message incorrectement formaté ou des modifications à des fichiers hors de leur zone réservée.

Application des poussées en avance rapide

Il ne reste plus qu'à forcer les poussées en avance rapide uniquement. À partir de la version 1.6, les paramètres `receive.denyDeletes` et `receive.denyNonFastForwards` règlent le problème. Cependant, l'utilisation d'un crochet permet de fonctionner avec des versions antérieures de Git et même après modification, des permissions par utilisateur ou toute autre évolution.

L'algorithme consiste à vérifier s'il y a des commits accessibles depuis l'ancienne révision qui ne sont pas accessibles depuis la nouvelle. S'il n'y en a aucun alors la poussée est effectivement en avance rapide. Sinon, il faut le rejeter :

```
# Forcer les poussées qu'en avance rapide
def verif_avance_rapide
  refs_manquees = `git rev-list #{$nouvellerev}..#{$anciennerev}`
  nb_refs_manquees = refs_manquees.split("\n").size
  if nb_refs_manquees > 0
    puts "[REGLE] Poussée en avance rapide uniquement"
    exit 1
  end
end

verif_avance_rapide
```

Tout est en place. En lançant `chmod u+x .git/hooks/update`, `update` étant le fichier dans lequel tout le code précédent réside, puis en essayant de pousser une référence qui n'est pas en avance rapide, on obtient ceci :

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Vérification des règles...
(refs/heads/master) (8338c5) (c5b616)
[REGLE] Poussée en avance rapide uniquement
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Il y a plusieurs point à relever ici. Premièrement, une ligne indique l'endroit où le crochet est appelé.

```
Vérification des règles...
(refs/heads/master) (fb8c72) (c56860)
```

Le script update affiche ces lignes sur stdout au tout début. Tout ce que le script écrit sur stdout sera transmis au client.

La ligne suivante à remarquer est le message d'erreur.

```
[REGLE] Poussée en avance rapide uniquement
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Le première ligne a été écrite par le script, les deux autres l'ont été par Git pour indiquer que le script `update` a rendu un code de sortie non nul, ce qui a causé l'échec de la poussée. Enfin, il y a ces lignes :

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Il y a un message d'échec distant pour chaque référence que le crochet a rejetée et une indication que l'échec est dû spécifiquement à un échec du crochet.

Par ailleurs, si la marque ref n'est pas présente dans le message de validation, le message d'erreur spécifique est affiché :

```
[REGLE] Le message de validation ne suit pas le format
```

Ou si quelqu'un cherche à modifier un fichier auquel il n'a pas les droits d'accès lors d'une poussée, il verra quelque chose de similaire. Par exemple, si un auteur de documentation essaie de pousser un commit qui modifie quelque chose dans le répertoire `lib`, il verra

```
[ACL] Vous n'avez pas le droit de pousser sur lib/test.rb
```

C'est tout. À partir de maintenant, tant que le script `update` est en place et exécutable, votre dépôt ne peut plus subir de poussées hors avancée rapide, n'accepte plus de messages sans format et vos utilisateurs sont bridés.

7.4.2 Crochets côté client

Le problème de cette approche, ce sont les plaintes des utilisateurs qui résulteront inévitablement des échecs de leurs poussées. Leur frustration et leur confusion devant le rejet à la dernière minute d'un travail minutieux est tout à fait compréhensible. De plus, la correction nécessitera une modification de leur historique, ce qui n'est pas une partie de plaisir.

Pour éviter ce scénario, il faut pouvoir fournir aux utilisateurs des crochets côté client qui leur permettront de vérifier que leurs validations seront effectivement acceptées par le serveur. Ainsi, ils pourront corriger les problèmes avant de valider et avant que ces difficultés ne deviennent des casse-têtes. Ces scripts n'étant pas diffusés lors du clonage du projet, il vous faudra les distribuer d'une autre manière, puis indiquer aux utilisateurs de les copier dans leur répertoire `.git/hooks` et de les rendre exécutables. Vous pouvez distribuer ces crochets au sein du projet ou dans un projet annexe mais il n'y a aucun moyen de les mettre en place automatiquement.

Premièrement, pour éviter le rejet du serveur au motif d'un mauvais format du message de validation, il faut vérifier celui-ci avant que chaque commit ne soit enregistré. Pour ce faire, utilisons le crochet `commit-msg`. En lisant le message à partir du fichier passé en premier argument et en le comparant au format attendu, on peut forcer Git à abandonner la validation en cas d'absence de correspondance :

```
#!/usr/bin/env ruby
fichier_message = ARGV[0]
message = File.read(fichier_message)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[REGLE] Le message de validation ne suit pas le format"
  exit 1
end
```

Avec ce fichier exécutable et à sa place dans `.git/hooks/commit-msg`, si une validation avec un message incorrect est tentée, voici le résultat :

```
$ git commit -am 'test'
[REGLE] Le message de validation ne suit pas le format
```

La validation n'a pas abouti. Néanmoins, si le message contient la bonne forme, Git accepte la validation :

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Ensuite, il faut s'assurer des droits sur les fichiers modifiés. Si le répertoire `.git` du projet contient une copie du fichier d'ACL précédemment utilisé, alors le script `pre-commit` suivant appliquera ses règles :

```
#!/usr/bin/env ruby

$utilisateur    = ENV['USER']

# [ insérer la fonction acl_access_data method ci-dessus ]

# Ne permet qu'à certains utilisateurs de modifier certains sous-répertoires
def verif_perms_repertoire
  acces = get_acl_access_data('.git/acl')

  fichiers_modifies = `git diff-index --cached --name-only HEAD`.split("\n")
  fichiers_modifies.each do |chemin|
    next if chemin.size == 0
    acces_permis = false
    acces[$utilisateur].each do |chemin_acces|
    if !chemin_acces || (chemin.index(chemin_acces) == 0)
      acces_permis = true
    end
    if !acces_permis
      puts "[ACL] Vous n'avez pas le droit de pousser sur #{path}"
      exit 1
    end
  end
end

verif_perms_repertoire
```

C'est grossièrement le même script que celui côté serveur, mais avec deux différences majeures. Premièrement, le fichier ACL est à un endroit différent parce que le script s'exécute depuis le copie de travail et non depuis le répertoire Git. Il faut donc changer le chemin vers le fichier d'ACL de

```
acces = get_acl_access_data('acl')
```

pour

```
acces = get_acl_access_data('.git/acl')
```

L'autre différence majeure réside dans la manière d'obtenir la liste des fichiers modifiés. La fonction sur le serveur la recherche dans le journal des commits mais comme dans le cas actuel, le commit n'a pas encore été enregistré, il faut chercher la liste dans la zone d'index. Donc au lieu de

```
fichiers_modifies = `git log -1 --name-only --pretty=format:'' #{ref}`
```

on utilise

```
fichiers_modifies = `git diff-index --cached --name-only HEAD`
```

Mais à ces deux différences près, le script fonctionne de manière identique. Ce script a aussi une autre limitation : il s'attend à ce que l'utilisateur qui le lance localement soit identique à celui sur le serveur distant. S'ils sont différents, il faudra positionner manuellement la variable `$utilisateur`.

La dernière action à réaliser consiste à vérifier que les références poussées sont bien en avance rapide, mais l'inverse est plutôt rare. Pour obtenir une référence qui n'est pas en avance rapide, il faut soit rebaser après un commit qui a déjà été poussé, soit essayer de pousser une branche locale différente vers la même branche distante.

Comme le serveur indiquera qu'on ne peut pas pousser sans avance rapide de toute façon et que le crochet empêche les poussées forcées, la seule action accidentelle qu'il faut intercepter reste le rebasage de commits qui ont déjà été poussés.

Voici un exemple de script `pre-rebase` qui fait cette vérification. Ce script récupère une liste de tous les commits qu'on est sur le point de réécrire et vérifie s'ils existent dans une référence distante. S'il en trouve un accessible depuis une des références distantes, il interrompt le rebasage :

```
#!/usr/bin/env ruby

branche_base = ARGV[0]
if ARGV[1]
  branche_thematique = ARGV[1]
else
  branche_thematique = "HEAD"
end

sha_cibles = `git rev-list #{branche_base}..#{branche_thematique}`.split("\n")
refs_distantes = `git branch -r`.split("\n").map { |r| r.strip }

shas_cibles.each do |sha|
  refs_distantes.each do |ref_distante|
    shas_pousses = `git rev-list ^#{sha}^@ refs/remotes/#{ref_distante}`
    if shas_pousses.split("\n").include?(sha)
      puts "[REGLE] Le commit #{sha} a déjà été poussé sur #{ref_distante}"
      exit 1
    end
  end
end
```

Ce script utilise une syntaxe qui n'a pas été abordée à la section « sélection de révision » du chapitre 6. La liste des commits déjà poussés est obtenue avec cette commande :

```
git rev-list ^#{sha}^@ refs/remotes/#{ref_distante}
```

La syntaxe `SHA^@` fait référence à tous le parents du commit. Les commits recherchés sont accessibles depuis le dernier commit distant et inaccessibles depuis n'importe quel parent de n'importe quel SHA qu'on cherche à pousser. C'est la définition d'avance rapide.

La limitation de cette approche reste qu'elle peut s'avérer très lente et non nécessaire. Si vous n'essayez pas de forcer à pousser avec l'option `-f`, le serveur vous avertira et n'acceptera pas la poussée. Cependant, cela reste un exercice intéressant qui peut aider théoriquement à éviter un rebasage qui devra être annulé plus tard.

7.5 Résumé

Nous avons traité la plupart des moyens principaux de personnaliser le client et le serveur Git pour mieux l'adapter à toutes les méthodes et les projets. Nous avons couvert toutes sortes de réglages de configurations, d'attributs dans des fichiers et de crochets d'évènement et nous avons construit un exemple de politique de gestion de serveur. Vous voilà prêt à régler Git à s'adapter à quasiment toutes les gestions dont vous avez rêvé.

Chapitre 8

Git et les autres systèmes

Le monde n'est pas parfait. Habituellement, vous ne pouvez pas basculer immédiatement sous Git tous les projets que vous pourriez rencontrer. Quelques fois, vous êtes bloqué sur un projet utilisant un autre VCS et très souvent ce système s'avère être Subversion. Dans la première partie de ce chapitre, nous traiterons de `git svn`, la passerelle bidirectionnelle de Git pour Subversion.

À un moment, vous voudrez convertir votre projet à Git. La seconde partie de ce chapitre traite la migration de votre projet dans Git : depuis Subversion, puis depuis Perforce et enfin par un script d'import personnalisé pour les cas non-standards.

8.1 Git et Subversion

Aujourd'hui, la majorité des projets de développement libre et un grand nombre de projets dans les sociétés utilisent Subversion pour gérer leur code source. C'est le VCS libre le plus populaire depuis une bonne décennie. Il est aussi très similaire à CVS qui a été le grand chef des gestionnaires de source avant lui.

Une des grandes fonctionnalités de Git est sa passerelle vers subversion, `git svn`. Cet outil vous permet d'utiliser Git comme un client valide d'un serveur Subversion pour que vous puissiez utiliser les capacités de Git en local puis poussez sur le serveur Subversion comme si vous utilisiez Subversion localement. Cela signifie que vous pouvez réaliser localement les embranchements et les fusions, utiliser l'index, utiliser le rebasage et la sélection de commits, etc, tandis que vos collaborateurs continuent de travailler avec leurs méthodes ancestrales et obscures. C'est une bonne manière d'introduire Git dans un environnement professionnel et d'aider vos collègues développeurs à devenir plus efficaces tandis que vous ferez pression pour une modification de l'infrastructure vers l'utilisation massive de Git. La passerelle Subversion n'est que la première dose vers la drogue du monde des DVCS.

8.1.1 git svn

La commande de base dans Git pour toutes les commandes de passerelle est `git svn`. Vous préposerez tout avec cette paire de mots. Les possibilités étant nombreuses, nous traiterons des plus communes pendant que nous détaillerons quelques petits modes de gestion.

Il est important de noter que lorsque vous utilisez `git svn`, vous interagissez avec Subversion qui est un système bien moins sophistiqué que Git. Bien que vous puissiez simplement réaliser des branches locales et les fusionner, il est généralement conseillé de conserver votre historique le plus

linéaire possible en rebasant votre travail et en évitant des activités telles que d'interagir dans le même temps avec un dépôt Git distant.

Ne réécrivez pas votre historique avant d'essayer de pousser à nouveau et ne poussez pas en parallèle dans un dépôt Git pour collaborer avec vos collègues développant avec Git. Subversion ne supporte qu'un historique linéaire et l'égarer est très facile. Si vous travaillez avec une équipe dont certains membres utilisent svn et d'autres utilisent Git, assurez-vous que tout le monde n'utilise que le serveur svn pour collaborer, cela vous rendra service.

8.1.2 Installation

Pour montrer cette fonctionnalité, il faut un serveur svn sur lequel vous avez des droits en écriture. Pour copier ces exemples, faites une copie inscriptible de mon dépôt de test. Dans cette optique, vous pouvez utiliser un outil appelé `svnsync` qui est livré avec les versions les plus récentes de Subversion — il devrait être distribué avec les versions à partir de 1.4. Pour ces tests, j'ai créé sur Google code un nouveau dépôt Subversion qui était une copie partielle du projet `protobuf` qui est un outil qui encode les données structurées pour une transmission par réseau.

En préparation, créez un nouveau dépôt local Subversion :

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Ensuite, autorisez tous les utilisateurs à changer les revprops — le moyen le plus simple consiste à ajouter un script pre-revprop-change que rend toujours 0 :

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Vous pouvez à présent synchroniser ce projet sur votre machine locale en lançant `svnsync init` avec les dépôts sources et cibles.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Cela initialise les propriétés nécessaires à la synchronisation. Vous pouvez ensuite cloner le code en lançant

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Bien que cette opération ne dure que quelques minutes, si vous essayez de copier le dépôt original sur un autre dépôt distant au lieu d'un dépôt local, le processus durera près d'une heure, en dépit du fait qu'il y a moins de 100 commits. Subversion doit cloner révision par révision puis pousser vers un autre dépôt — c'est ridiculement inefficace mais c'est la seule possibilité.

8.1.3 Démarrage

Avec des droits en écriture sur un dépôt Subversion, vous voici prêt à expérimenter une méthode typique. Commençons par la commande `git svn clone` qui importe un dépôt Subversion complet dans un dépôt Git local. Souvenez-vous que si vous importez depuis un dépôt Subversion hébergé sur internet, il faut remplacer l'URL `file://tmp/test-svn` ci-dessous par l'URL de votre dépôt Subversion :

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
      A    m4/acx_pthread.m4
      A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
 file:///tmp/test-svn/branches/my-calc-branch r76
```

Cela équivaut à lancer `git svn init` suivi de `git svn fetch` sur l'URL que vous avez fournie. Cela peut prendre un certain temps. Le projet de test ne contient que 75 commits et la taille du code n'est pas extraordinaire, ce qui prend juste quelques minutes. Cependant, Git doit extraire chaque version, une par une et les valider individuellement. Pour un projet contenant des centaines ou des milliers de commits, cela peut prendre littéralement des heures ou même des jours à terminer.

La partie `-T trunk -b branches -t tags` indique à Git que ce dépôt Subversion suit les conventions de base en matière d'embranchement et de balisage. Si vous nommez votre trunk, vos branches ou vos balises différemment, vous pouvez modifier ces options. Comme cette organisation est la plus commune, ces options peuvent être simplement remplacées par `-s` qui signifie structure standard. La commande suivante est équivalente :

```
$ git svn clone file:///tmp/test-svn -s
```

À présent, vous disposez d'un dépôt Git valide qui a importé vos branches et vos balises :

```
$ git branch -a
* master
```

```
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

Il est important de remarquer comment cet outil sous-classe vos références distantes différemment. Quand vous clonez un dépôt Git normal, vous obtenez toutes les branches distantes localement sous la forme `origin/[branch]` avec un espace de nom correspondant au dépôt distant. Cependant, `git svn` assume que vous n'aurez pas de multiples dépôts distants et enregistre toutes ses références pour qu'elles pointent sur le dépôt distant. Cependant, vous pouvez utiliser la commande Git de plomberie `show-ref` pour visualiser toutes vos références.

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

Pour un dépôt Git normal, cela ressemble plus à ceci :

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

Ici, vous disposez de deux serveurs distants : un nommé `gitserver` avec une branche `master` et un autre nommé `origin` avec deux branches `master` et `testing`.

Remarquez comme dans cet exemple de références distantes importées via `git svn`, les balises sont ajoutées comme des branches distantes et non comme des vraies balises Git. Votre importation Subversion indique plutôt qu'il a un serveur distant appelé `tags` présentant des branches.

8.1.4 Valider en retour sur le serveur Subversion

Comme vous disposez d'un dépôt en état de marche, vous pouvez commencer à travailler sur le projet et pousser vos commits en utilisant efficacement Git comme client SVN. Si vous éditez un des fichiers et le validez, vous créez un commit qui existe localement dans Git mais qui n'existe pas sur le serveur Subversion :

```
$ git commit -am 'Ajout d'instructions pour git-svn dans LISEZMOI'
[master 97031e5] Ajout d'instructions pour git-svn dans LISEZMOI
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Ensuite, vous avez besoin de pousser vos modifications en amont. Remarquez que cela modifie la manière de travailler par rapport à Subversion — vous pouvez réalisez plusieurs validations en mode déconnecté pour ensuite les pousser toutes en une fois sur le serveur Subversion. Pour pousser sur un serveur Subversion, il faut lancer la commande `git svn dcommit` :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r79
       M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Cette commande rassemble tous les commits que vous avez validés par dessus le code du serveur Subversion et réalise un commit sur le serveur pour chacun, puis réécrit l'historique Git local pour y ajouter un identifiant unique. Cette étape est à souligner car elle signifie que toutes les sommes de contrôle SHA-1 de vos commits locaux ont changé. C'est en partie pour cette raison que c'est une idée très périlleuse de vouloir travailler dans le même temps avec des serveurs Git distants. L'examen du dernier commit montre que le nouveau `git-svn-id` a été ajouté :

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Ajout d'instructions pour git-svn dans LISEZMOI

    git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

Remarquez que la somme de contrôle SHA qui commençait par `97031e5` quand vous avez validé commence à présent par `938b1a5`. Si vous souhaitez pousser à la fois sur un serveur Git et un serveur Subversion, il faut obligatoirement pousser (`dcommit`) sur le serveur Subversion en premier, car cette action va modifier vos données des commits.

8.1.5 Tirer des modifications

Quand vous travaillez avec d'autres développeurs, il arrive à certains moments que ce qu'un développeur a poussé provoque un conflit lorsqu'un autre voudra pousser à son tour. Cette modification sera rejetée jusqu'à ce qu'elle soit fusionnée. Dans `git svn`, cela ressemble à ceci :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

Pour résoudre cette situation, vous pouvez lancer la commande `git svn rebase` qui tire depuis le serveur toute modification apparue entre temps et rebase votre travail sur le sommet de l'historique du serveur :

```
$ git svn rebase
       M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

À présent, tout votre travail se trouve au delà de l'historique du serveur et vous pouvez effectivement réaliser un `dcommit` :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r81
       M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Il est important de se souvenir qu'à la différence de Git qui requiert une fusion avec les modifications distantes non présentes localement avant de pouvoir pousser, `git svn` ne vous y contraint que si vos modifications provoquent un conflit. Si une autre personne pousse une modification à un fichier et que vous poussez une modification à un autre fichier, votre `dcommit` passera sans problème :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      configure.ac
Committed r84
       M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
       M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M   autogen.sh
```

```
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Il faut s'en souvenir car le résultat de ces actions est un état du dépôt qui n'existait pas sur aucun des ordinateurs quand vous avez poussé. Si les modifications sont incompatibles mais ne créent pas de conflits, vous pouvez créer des défauts qui seront très difficiles à diagnostiquer. C'est une grande différence avec un serveur Git — dans Git, vous pouvez tester complètement l'état du projet sur votre système client avant de le publier, tandis qu'avec SVN, vous ne pouvez jamais être totalement certain que les états avant et après validation sont identiques.

Vous devrez aussi lancer cette commande pour tirer les modifications depuis le serveur Subversion, même si vous n'êtes pas encore prêt à valider. Vous pouvez lancer `git svn fetch` pour tirer les nouveaux commits, mais `git svn rebase` tire non seulement les commits distants mais rebase aussi vos commit locaux.

```
$ git svn rebase
       M      generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Lancer `git svn rebase` de temps en temps vous assure que votre travail est toujours synchronisé avec le serveur. Vous devrez cependant vous assurer que votre copie de travail est propre quand vous la lancez. Si vous avez des modifications locales, il vous faudra soit remiser votre travail, soit valider temporairement vos modifications avant de lancer `git svn rebase`, sinon la commande s'arrêtera si elle détecte que le rebasage provoquerait un conflit de fusion.

8.1.6 Le problème avec les branches Git

Après vous être habitué à la manière de faire avec Git, vous souhaiterez sûrement créer des branches thématiques, travailler dessus, puis les fusionner. Si vous poussez sur un serveur Subversion via git svn, vous souhaiterez à chaque fois rebaser votre travail sur une branche unique au lieu de fusionner les branches ensemble. La raison principale en est que Subversion gère un historique linéaire et ne gère pas les fusions comme Git y excelle. De ce fait, git svn suit seulement le premier parent lorsqu'il convertit les instantanés en commits Subversion.

Supposons que votre historique ressemble à ce qui suit. Vous avez créé une branche `experience`, avez réalisé deux validations puis les avez fusionnées dans master. Lors du `dcommit`, vous voyez le résultat suivant :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      CHANGES.txt
Committed r85
       M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
```

```
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
       M      COPYING.txt
       M      INSTALL.txt
Committed r86
       M      INSTALL.txt
       M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Lancer `dcommit` sur une branche avec un historique fusionné fonctionne correctement, à l'exception que l'examen de l'historique du projet Git indique qu'il n'a réécrit aucun des commits réalisés sur la branche `experience`, mais que toutes les modifications introduites apparaissent dans la version SVN de l'unique commit de fusion.

Quand quelqu'un d'autre clone ce travail, tout ce qu'il voit, c'est le commit de la fusion avec toutes les modifications injectées en une fois. Il ne voit aucune information sur son origine ni sur sa date de validation.

8.1.7 Les embranchements dans Subversion

La gestion de branches dans Subversion n'a rien à voir avec celle de Git. Évitez de l'utiliser tant que possible. Cependant vous pouvez créer des branches et valider dessus dans Subversion en utilisant git svn.

Créer une nouvelle branche SVN

Pour créer une nouvelle branche dans Subversion, vous pouvez utiliser la commande `git svn branch [nom de la branche]` :

```
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Cela est équivalent à la commande Subversion `svn copy trunk branches/opera` et réalise l'opération sur le serveur Subversion. Remarquez que cette commande ne vous bascule pas sur cette branche ; si vous validez, le commit s'appliquera à `trunk` et non à la branche `opera`.

8.1.8 Basculer de branche active

Git devine la branche cible des dcommits en se référant au sommet des branches Subversion dans votre historique — vous ne devriez en avoir qu'un et celui-ci devrait être le dernier possédant un `git-svn-id` dans l'historique actuel de votre branche.

Si vous souhaitez travailler simultanément sur plusieurs branches, vous pouvez régler vos branches locales pour que le `dcommit` arrive sur une branche Subversion spécifique en les démarrant sur le commit de cette branche importée depuis Subversion. Si vous voulez une branche `opera` sur laquelle travailler séparément, vous pouvez lancer

```
$ git branch opera remotes/opera
```

À présent, si vous voulez fusionner votre branche `opera` dans `trunk` (votre branche `master`), vous pouvez le faire en réalisant un `git merge` normal. Mais vous devez préciser un message de validation descriptif (via `-m`), ou la fusion indiquera simplement « Merge branch opera » au lieu d'un message plus informatif.

Souvenez-vous que bien que vous utilisez `git merge` qui facilitera l'opération de fusion par rapport à Subversion (Git détectera automatiquement l'ancêtre commun pour la fusion), ce n'est pas un commit de fusion normal de Git. Vous devrez pousser ces données finalement sur le serveur Subversion qui ne sait pas tracer les commits possédant plusieurs parents. Donc, ce sera un commit unique qui englobera toutes les modifications de l'autre branche. Après avoir fusionné une branche dans une autre, il est difficile de continuer à travailler sur cette branche, comme vous le feriez normalement dans Git. La commande `dcommit` qui a été lancée efface toute information sur la branche qui a été fusionnée, ce qui rend faux tout calcul d'antériorité pour la fusion. `dcommit` fait ressembler le résultat de `git merge` à celui de `git merge --squash`. Malheureusement, il n'y a pas de moyen efficace de remédier à ce problème — Subversion ne stocke pas cette information et vous serez toujours contraints par ses limitations si vous l'utilisez comme serveur. Pour éviter ces problèmes, le mieux reste d'effacer la branche locale (dans notre cas, `opera`) dès qu'elle a été fusionnée dans `trunk`.

8.1.9 Les commandes Subversion

La boîte à outil `git svn` fournit des commandes de nature à faciliter la transition vers Git en mimant certaines commandes disponibles avec Subversion. Voici quelques commandes qui vous fournissent les mêmes services que Subversion.

L'historique dans le style Subversion

Si vous êtes habitué à Subversion, vous pouvez lancer `git svn log` pour visualiser votre historique dans un format SVN :

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'
```

```
------------------------------------------------------------------------
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```

Deux choses importantes à connaître sur `git svn log` : premièrement, à la différence de la commande réelle `svn log` qui interroge le serveur, cette commande fonctionne hors connexion ; deuxièmement, elle ne montre que les commits qui ont été effectivement remontés sur le serveur Subversion. Les commits locaux qui n'ont pas encore été remontés via `dcommit` n'apparaissent pas, pas plus que ceux qui auraient été poussés sur le serveur par des tiers entre deux `git svn rebase`. Cela donne plutôt le dernier état connu des commits sur le serveur Subversion.

Les annotations SVN

De la même manière que `git svn log` simule une commande `svn log` déconnectée, vous pouvez obtenir l'équivalent de `svn annotate` en lançant `git svn blame [FICHIER]`. Le résultat ressemble à ceci :

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Ici aussi, tous les commits locaux dans Git ou ceux poussé sur Subversion dans l'intervalle n'apparaissent pas.

L'information sur la serveur SVN

Vous pouvez aussi obtenir le même genre d'information que celle fournie par `svn info` en lançant `git svn info` :

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
```

```
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Comme `blame` et `log`, cette commande travaille hors connexion et n'est à jour qu'à la dernière date à laquelle vous avez communiqué avec le serveur Subversion.

Ignorer ce que Subversion ignore

Si vous clonez un dépôt Subversion contenant des propriétés `svn:ignore`, vous souhaiterez sûrement paramétrer les fichiers `.gitignore` en correspondance pour vous éviter de valider accidentellement des fichiers interdits. `git svn` dispose de deux commandes pour le faire.

La première est `git svn create-ignore` qui crée automatiquement pour vous les fichiers `.gitignore` prêts pour l'inclusion dans votre prochaine validation.

La seconde commande est `git svn show-ignore` qui affiche sur stdout les lignes nécessaires à un fichier `.gitignore` qu'il suffira de rediriger dans votre fichier d'exclusion de projet :

```
$ git svn show-ignore > .git/info/exclude
```

De cette manière, vous ne parsemez pas le projet de fichiers `.gitignore`. C'est une option optimale si vous êtes le seul utilisateur de Git dans une équipe Subversion et que vos coéquipiers ne veulent pas voir de fichiers `.gitignore` dans le projet.

8.1.10 Résumé sur Git-Svn

Les outils `git svn` sont utiles si vous êtes bloqué avec un serveur Subversion pour le moment ou si vous devez travailler dans un environnement de développement qui nécessite un serveur Subversion. Il faut cependant les considérer comme une version tronquée de Git ou vous pourriez rencontrer des problèmes de conversion synonymes de troubles pour vous et vos collaborateurs. Pour éviter tout problème, essayez de suivre les principes suivants :

- Garder un historique Git linéaire qui ne contient pas de commits de fusion issus de `git merge`. Rebasez tout travail réalisé en dehors de la branche principale sur celle-ci ; ne la fusionnez pas.

- Ne mettez pas en place et ne travaillez pas en parallèle sur un serveur Git. Si nécessaire, montez-en un pour accélérer les clones pour de nouveaux développeurs mais n'y poussez rien qui n'ait déjà une entrée `git-svn-id`. Vous devriez même y ajouter un crochet `pre-receive` qui vérifie la présence de `git-svn-id` dans chaque message de validation et rejette les remontées dont un des commits n'en contiendrait pas.

Si vous suivez ces principes, le travail avec un serveur Subversion peut être supportable. Cependant, si le basculement sur un vrai serveur Git est possible, votre équipe y gagnera beaucoup.

8.2 Migrer sur Git

Si vous avez une base de code dans un autre VCS et que vous avez décidé d'utiliser Git, vous devez migrer votre projet d'une manière ou d'une autre. Ce chapitre traite d'outils d'import inclus dans Git avec des systèmes communs et démontre comment développer votre propre outil.

8.2.1 Importer

Nous allons détailler la manière d'importer des données à partir de deux des plus grands systèmes SCM utilisés en milieu professionnel, Subversion et Perforce, pour les raisons combinées qu'ils regroupent la majorité des utilisateurs que je connais migrer vers Git et que des outils de grande qualité pour ces deux systèmes sont distribués avec Git.

8.2.2 Subversion

Si vous avez lu la section précédente sur l'utilisation de `git svn`, vous pouvez facilement utiliser ces instructions pour réaliser un `git svn clone` du dépôt. Ensuite, arrêtez d'utiliser le serveur Subversion, poussez sur un nouveau serveur Git et commencez à l'utiliser. Si vous voulez l'historique, vous pouvez l'obtenir aussi rapidement que vous pourrez tirer les données du serveur Subversion (ce qui peut prendre un certain temps).

Cependant, l'import n'est pas parfait ; et comme cela prend autant de temps, autant le faire bien. Le premier problème est l'information d'auteur. Dans Subversion, chaque personne qui valide dispose d'un compte sur le système qui est enregistré dans l'information de validation. Les exemples de la section précédente montrent `schacon` à certains endroits, tels que la sortie de `blame` ou de `git svn log`. Si vous voulez transposer ces données vers des données d'auteur au format Git, vous avez besoin d'une correspondance entre les utilisateurs Subversion et les auteurs Git. Créez un fichier appelé `users.txt` contenant cette équivalence dans le format suivant :

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Pour récupérer la liste des noms d'auteurs utilisés par SVN, vous pouvez utiliser la ligne suivante :

```
$ svn log --xml | grep author | sort -u | perl -pe 's/.>(.?)<./$1 = /'
```

Cela génère une sortie au format XML — vous pouvez visualiser les auteurs, créer une liste unique puis éliminer l'XML. Évidemment, cette ligne ne fonctionne que sur une machine disposant des commandes `grep`, `sort` et `perl`. Ensuite, redirigez votre sortie dans votre fichier users.txt pour pouvoir y ajouter en correspondance les données équivalentes Git.

Vous pouvez alors fournir ce fichier à `git svn` pour l'aider à convertir les données d'auteur plus précisément. Vous pouvez aussi indiquer à `git svn` de ne pas inclure les méta-données que Subversion importe habituellement en passant l'option `--no-metadata` à la commande `clone` ou `init`. Au final, votre commande d'import ressemble à ceci :

```
$ git-svn clone http://mon-projet.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Maintenant, l'import depuis Subversion dans le répertoire `my_project` est plus présentable. En lieu et place de commits qui ressemblent à ceci :

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

les commits ressemblent à ceci :

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Non seulement le champ auteur a meilleure mine, mais de plus le champ `git-svn-id` a disparu.

Il est encore nécessaire de faire un peu de ménage `post-import`. Déjà, vous devriez nettoyer les références bizarres que `git svn` crée. Premièrement, déplacez les balises pour qu'elles soient de vraies balises plutôt que des branches distantes étranges, ensuite déplacez le reste des branches pour qu'elles deviennent locales.

Pour déplacer les balises et en faire de vraies balises Git, lancez

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

Cela récupère les références déclarées comme branches distantes commençant par `tags/` et les transforme en vraies balises (légères).

Ensuite, déplacez le reste des références sous `refs/remotes` en branches locales :

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

À présent, toutes les vieilles branches sont des vraies branches Git et toutes les vieilles balises sont de vraies balises Git. La dernière activité consiste à ajouter votre nouveau serveur Git comme serveur distant et à y pousser votre projet transformé. Pour pousser tout, y compris branches et balises, lancez :

```
$ git push origin --all
```

Toutes vos données, branches et tags sont à présent disponibles sur le serveur Git comme import propre et naturel.

8.2.3 Perforce

L'autre système duquel on peut souhaiter importer les données est Perforce. Un outil d'import Perforce est aussi distribué avec Git, mais seulement dans la section `contrib` du code source. Il n'est pas disponible par défaut comme `git svn`. Pour le lancer, il vous faut récupérer le code source de Git que vous pouvez télécharger à partir de git.kernel.org :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

Dans ce répertoire `fast-import`, vous devriez trouver une script exécutable Python appelé `git-p4`. Python et l'outil `p4` doivent être installés sur votre machine pour que cet import fonctionne. Par exemple, nous importerons le projet Jam depuis le Perforce Public Depot. Pour installer votre client, vous devez exporter la variable d'environnement `P4PORT` qui pointe sur le dépôt Perforce :

```
$ export P4PORT=public.perforce.com:1666
```

Lancez la commande `git-p4 clone` pour importer la projet Jam depuis le serveur Perforce, en fournissant le dépôt avec le chemin du projet et le chemin dans lequel vous souhaitez importer le projet :

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

Si vous vous rendez dans le répertoire `/opt/p4import` et lancez la commande `git log`, vous pouvez examiner votre projet importé :

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Vous pouvez visualiser l'identifiant `git-p4` de chaque commit. Il n'y a pas de problème à garder cet identifiant ici, au cas où vous auriez besoin de référencer dans l'avenir le numéro de modification Perforce. Cependant, si vous souhaitez supprimer l'identifiant, c'est le bon moment, avant de commencer à travailler avec le nouveau dépôt. Vous pouvez utiliser `git filter-branch` pour faire une retrait en masse des chaînes d'identifiant :

```
$ git filter-branch --msg-filter '
        sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

Si vous lancez `git log`, vous vous rendez compte que toutes les sommes de contrôle SHA-1 des commits ont changé, mais aussi que plus aucune chaîne `git-p4` n'apparaît dans les messages de validation :

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800
```

```
    Update derived jamgram.c
```

Votre import est fin prêt pour être poussé sur un nouveau serveur Git.

8.2.4 Un outil d'import personnalisé

Si votre système n'est ni Subversion, ni Perforce, vous devriez rechercher sur Internet un outil d'import spécifique — il en existe de bonne qualité pour CVS, Clear Case, Visual Source Safe ou même pour un répertoire d'archives. Si aucun de ses outils ne fonctionne pour votre cas, que vous ayez un outil plus rare ou que vous ayez besoin d'un mode d'import personnalisé, `git fast-import` peut être la solution. Cette commande lit de simples instructions sur stdin pour écrire les données spécifiques Git. C'est tout de même plus simple pour créer les objets Git que de devoir utiliser les commandes Git brutes ou d'essayer d'écrire directement les objets (voir Chapitre 9 pour plus d'information). De cette façon, vous écrivez un script d'import qui lit les informations nécessaires depuis le système d'origine et affiche des instructions directes sur stdout. Vous pouvez alors simplement lancer ce programme et rediriger sa sortie dans `git fast-import`.

Pour démontrer rapidement cette fonctionnalité, nous allons écrire un script simple d'import. Supposons que vous travailliez dans `en_cours` et que vous fassiez des sauvegardes de temps en temps dans des répertoires nommés avec la date `back_AAAA_MM_JJ` et que vous souhaitiez importer ceci dans Git. Votre structure de répertoire ressemble à ceci :

```
$ ls /opt/import_depuis
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
en_cours
```

Pour importer un répertoire dans Git, vous devez savoir comment Git stocke ses données. Comme vous pouvez vous en souvenir, Git est à la base une liste chaînée d'objets de commits qui pointent sur un instantané de contenu. Tout ce qu'il y a à faire donc, et d'indiquer à `fast-import` ce que sont les instantanés de contenu, quelles données de commit pointent dessus et l'ordre dans lequel ils s'enchaînent. La stratégie consistera à parcourir les instantanés un par un et à créer des commits avec le contenu de chaque répertoire, en le reliant à son prédécesseur.

Comme déjà fait dans la section « Un exemple de règle appliquée par Git » du chapitre 7, nous l'écrirons en Ruby parce que c'est le langage avec lequel je travaille en général et qu'il est assez facile à lire. Vous pouvez facilement retranscrire cet exemple dans votre langage de prédilection, la seule contrainte étant qu'il doit pouvoir afficher les informations appropriées sur stdout. Si vous travaillez sous Windows, cela signifie que vous devrez faire particulièrement attention à ne pas introduire de retour chariot à la fin de vos lignes. `git fast-import` n'accepte particulièrement que les sauts de ligne (line feed LF) et pas les retour chariot saut de ligne (CRLF) utilisés par Windows.

Pour commencer, déplaçons nous dans le répertoire cible et identifions chaque sous-répertoire, chacun représentant un instantané que vous souhaitez importer en tant que commit. Nous visiterons chaque sous-répertoire et afficherons les commandes nécessaires à son export. La boucle principale ressemble à ceci :

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Dans chaque répertoire, nous lançons `print_export` qui prend le manifest et la marque de l'instantané précédent et retourne le manifest et la marque de l'actuel ; de cette manière, vous pouvez les chaîner correctement. « Marque » est le terme de `fast-import` pour nommer un identifiant que vous donnez à un commit. Au fur et à mesure de la création des commits, vous leur attribuez une marque individuelle qui pourra être utilisée pour y faire référence depuis d'autres commits. La première chose à faire dans `print_export` est donc de générer une marque à partir du nom du répertoire :

```
mark = convert_dir_to_mark(dir)
```

Cela sera réalisé en créant un tableau des répertoires et en utilisant l'indice comme marque, celle-ci devant être un nombre entier. Votre méthode ressemble à ceci :

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Après une représentation entière de votre commit, vous avez besoin d'une date pour les méta-données du commit. La date est présente dans le nom du répertoire, alors analysons-le. La ligne suivante du fichier `print_export` est donc

```
date = convert_dir_to_date(dir)
```

où `convert_dir_to_date` est défini comme

```
def convert_dir_to_date(dir)
  if dir == 'en_cours'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Elle retourne une nombre entier pour la date de chaque répertoire. La dernière partie des méta-informations nécessaires à chaque commit est l'information du validateur qui sera stockée en dur dans une variable globale :

```
$author = 'Scott Chacon <schacon@example.com>'
```

Nous voilà prêt à commencer à écrire les informations de commit du script d'import. La première information indique qu'on définit un objet commit et la branche sur laquelle il se trouve, suivi de la marque qui a été générée, l'information du validateur et le message de validation et enfin le commit précédent, s'il existe. Le code ressemble à ceci :

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Nous codons en dur le fuseau horaire (–0700) car c'est simple. Si vous importez depuis un autre système, vous devez spécifier le fuseau horaire comme un décalage. Le message de validation doit être exprimé dans un format spécial :

```
data (taille)\n(contenu)
```

Le format est composé du mot « data », la taille des données à lire, un caractère saut de ligne, et finalement les données. Ce format est réutilisé plus tard, alors autant créer une méthode auxiliaire, `export_data` :

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Il reste seulement à spécifier le contenu en fichiers de chaque instantané. C'est facile, car vous les avez dans le répertoire. Git va alors enregistrer de manière appropriée chaque instantané :

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Comme de nombreux systèmes conçoivent leurs révisions comme des modifications d'un commit à l'autre, fast-import accepte aussi avec chaque commit des commandes qui spécifient quels fichiers ont été ajoutés, effacés ou modifiés et ce que sont les nouveaux contenus. Vous pourriez calculer les différences entre chaque instantané et ne fournir que ces données, mais cela est plus complexe — vous pourriez tout aussi bien fournir à Git toutes les données et lui laisser faire le travail. Si c'est ce qui convient mieux à vos données, référez-vous à la page de manuel de `fast-import` pour savoir comment fournir les données de cette façon.

Le format pour lister le contenu d'un nouveau fichier ou spécifier le nouveau contenu d'un fichier modifié est comme suit :

```
M 644 inline chemin/du/fichier
data (taille)
(contenu du fichier)
```

Ici, 644 est le mode (si vous avez des fichiers executables, vous devez le détecter et spécifier plutôt 755), « inline » signifie que le contenu du fichier sera listé immédiatement après cette ligne. La méthode `inline_data` ressemble à ceci :

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Nous réutilisons la méthode `export_data` définie plus tôt, car c'est la même méthode que pour spécifier les données du message de validation.

La dernière chose à faire consiste à retourner le marque actuelle pour pouvoir la passer à la prochaine itération :

```
return mark
```

NOTE : si vous utilisez Windows, vous devrez vous assurer d'ajouter une étape supplémentaire. Comme mentionné auparavant, Windows utilise CRLF comme caractère de retour à la ligne tandis que git fast-import s'attend à LF. Pour contourner ce problème et satisfaire git fast-import, il faut forcer ruby à utiliser LF au lieu de CRLF :

```
$stdout.binmode
```

Et voilà. Si vous lancez ce script, vous obtiendrez un contenu qui ressemble à ceci :

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

Pour lancer l'outil d'import, redirigez cette sortie dans `git fast-import` alors que vous vous trouvez dans le répertoire Git dans lequel vous souhaitez importer. Vous pouvez créer un nouveau répertoire, puis l'initialiser avec `git init`, puis lancer votre script :

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           18 (         1 duplicates                  )
      blobs  :            7 (         1 duplicates          0 deltas)
      trees  :            6 (         0 duplicates          1 deltas)
      commits:            5 (         0 duplicates          0 deltas)
      tags   :            0 (         0 duplicates          0 deltas)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              3
Memory total:          2255 KiB
       pools:          2098 KiB
     objects:           156 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
```

```
pack_report: core.packedGitWindowSize =   33554432
pack_report: core.packedGitLimit      =  268435456
pack_report: pack_used_ctr            =          9
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =       1356 /       1356
---------------------------------------------------------------------
```

Comme vous pouvez le remarquer, lorsqu'il se termine avec succès, il affiche quelques statistiques sur ses réalisations. Dans ce cas, 18 objets ont été importés en 5 validations dans 1 branche. À présent, `git log` permet de visualiser le nouvel historique :

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from en_cours

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

Et voilà ! Un joli dépôt Git tout propre. Il est important de noter que rien n'a été extrait. Présentement, aucun fichier n'est présent dans votre copie de travail. Pour les avoir, vous devez réinitialiser votre branche sur `master` :

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from en_cours
$ ls
file.rb  lib
```

Vous pouvez faire bien plus avec l'outil `fast-import` — gérer différents modes, les données binaires, les branches multiples et la fusion, les balises, les indicateurs de progrès, et plus encore. Des exemples de scénarios plus complexes sont disponibles dans le répertoire `contrib/fast-import` du code source Git ; un des meilleurs est justement le script `git-p4` traité précédemment.

8.3 Résumé

Vous devriez être à l'aise à l'utilisation de Git avec Subversion ou pour l'import de quasiment toutes le sortes de dépôts dans un nouveau Git sans perdre de données. Le chapitre suivant traitera des structures internes de Git pour vous permettre d'en retailler chaque octet, si le besoin s'en fait sentir.

Chapitre 9

Les trippes de Git

Vous êtes peut-être arrivé à ce chapitre en en sautant certains ou après avoir parcouru tout le reste du livre. Dans tous les cas, c'est ici que l'on parle du fonctionnement interne et de la mise en œuvre de Git. Pour moi, leur apprentissage a été fondamental pour comprendre à quel point Git est utile et puissant, mais d'autres soutiennent que cela peut être source de confusion et être trop complexe pour les débutants. J'en ai donc fait le dernier chapitre de ce livre pour que vous puissiez le lire tôt ou tard lors de votre apprentissage. Je vous laisse le choix.

Maintenant que vous êtes ici, commençons. Tout d'abord, et même si ce n'est pas clair tout de suite, Git est fondamentalement un système de fichiers adressables par contenu (content-addressable filesystem) avec l'interface utilisateur d'un VCS au-dessus. Vous en apprendrez plus à ce sujet dans quelques instants.

Aux premiers jours de Git (surtout avant la version 1.5), l'interface utilisateur était beaucoup plus complexe, car elle était centrée sur le système de fichier plutôt que sur l'aspect VCS. Ces dernières années, l'interface utilisateur a été peaufinée jusqu'à devenir aussi cohérente et facile à utiliser que n'importe quel autre système. Pour beaucoup, l'image du Git des début avec son interface utilisateur complexe et difficile à apprendre est toujours présente. La couche système de fichiers adressables par contenu est vraiment géniale et j'en parlerai dans ce chapitre. Ensuite, vous apprendrez les mécanismes de transport/transmission/communication ainsi que les tâches que vous serez amenées à faire pour maintenir un dépôt.

9.1 Plomberie et porcelaine

Ce livre couvre l'utilisation de Git avec une trentaine de verbes comme `checkout`, `branch`, `remote` … Mais, puisque Git était initialement une boîte à outils (N.d.T : Toolkit) pour VCS, plutôt d'un VCS complet et conviviale, il dispose de tout un ensemble d'action pour les tâches bas niveau qui étaient conçues pour être liées à la UNIX ou appelées depuis de scripts. Ces commandes sont dites commandes de « plomberie » (N.d.T « plumbing »), et les autres, plus conviviales sont appelées « porcelaines » (N.d.T : « porcelain »).

Les huit premiers chapitres du livre concernent presque exclusivement les commandes porcelaine. Par contre, dans ce chapitre, vous serez principalement confrontés aux commandes de plomberie bas niveaux, car elles vous donnent accès au fonctionnement interne de Git et aident à montrer comment et pourquoi Git fonctionne comme il le fait. Ces commandes ne sont pas faites pour être utilisées à la main sur ligne de commandes, mais sont plutôt utilisées comme briques de bases pour écrire de nouveaux outils et scripts personnalisés.

Quand vous exécutez `git init` dans un nouveau répertoire ou un répertoire existant, Git crée un répertoire `.git` qui contient presque tout ce que Git stocke et manipule. Si vous voulez sauvegarder ou cloner votre dépôt, copier ce seul répertoire suffirait presque. Ce chapitre traite principalement de ce que contient ce répertoire. Voici à quoi il ressemble :

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

Vous y verrez sans doute d'autres fichiers, mais ceci est un dépôt qui vient d'être crée avec `git init`, et c'est ce que vous verrez par défaut. Le répertoire `branches` n'est pas utilisé par les versions récentes de Git, et le fichier `description` est utilisé uniquement par le programme GitWeb, il ne faut donc pas s'en soucier. Le fichier `config` contient les options de configuration spécifiques à votre projet, et le répertoire `info` contient un fichier listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans un fichier .gitignore. Le répertoire `hooks` contient les scripts de procédures automatiques côté client ou serveur, ils sont décrits en détail dans le chapitre 6.

Il reste quatre éléments importants : les fichiers `HEAD` et `index`, ainsi que les répertoires `objects` et `refs`. Ce sont les parties centrales de Git. Le répertoire `objects` stocke le contenu de votre base de données, le répertoire `refs` stocke les pointeurs vers les objets commit de ces données (branches), le fichier `HEAD` pointe sur la branche qui est en cours dans votre répertoire de travail (checkout), et le fichier `index` est l'endroit où Git stocke les informations sur la zone d'attente. Vous allez maintenant plonger en détail dans chacune de ces sections et voir comment Git fonctionne.

9.2 Les objets Git

Git est un système de fichier adressables par contenu. Super! Mais qu'est-ce que ça veut dire? Ça veut dire que le cœur de Git est une simple base de paire clef/valeur. Vous pouvez y insérer n'importe quelle sorte de données, et il vous retournera une clé que vous pourrez utiliser à n'importe quel moment pour récupérer ces données à nouveau. Pour illustrer cela, vous pouvez utiliser la commande de plomberie `hash-object`, qui prend des données, les stocke dans votre répertoire `.git`, puis retourne la clé sous laquelle les données sont stockées. Tout d'abord, créez un nouveau dépôt Git et vérifier que rien ne se trouve dans le répertoire `object` :

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
```

```
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git a initialisé le répertoire `objects` et y a crée les sous-répertoires `pack` et `info`, mais ils ne contiennent pas de fichier régulier. Maintenant, stockez du texte dans votre base de données Git :

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

L'option `-w` spécifie à `hash-object` de stocker l'objet, sinon la commande répondrait seulement quelle serait la clé. `--stdin` spécifie à la commande de lire le contenu depuis l'entrée standard, sinon `hash-object` s'attend à trouver un chemin vers un fichier. La sortie de la commande est une empreinte de 40 caractères. C'est l'empreinte SHA-1 : une somme de contrôle du contenu du fichier que vous stockez plus une en-tête, dont les détails sont un peu plus bas. Voyez maintenant comment Git a stocké vos données :

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vous pouvez voir un fichier dans le répertoire `objects`. C'est comme cela que Git stocke initialement du contenu : un fichier par contenu, nommé d'après la somme de contrôle SHA-1 du contenu et de son en-tête. Le sous-répertoire est nommé d'après les 2 premiers caractères de l'empreinte et le fichier d'après les 38 caractères restants.

Vous pouvez récupérer le contenu avec la commande `cat-file`. Cette commande est un peu le couteau suisse pour l'inspection des objets Git. Utiliser `-p` avec `cat-file` vous permet de connaître le type de contenu et de l'afficher clairement :

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Vous pouvez maintenant ajouter du contenu à Git et le récupérer. Vous pouvez aussi faire ceci avec des fichiers. Par exemple, vous pouvez mettre en œuvre une gestion de version simple d'un fichier. D'abord, créez un nouveau fichier et enregistrez son contenu dans la base de données :

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Puis, modifiez le contenu du fichier, et enregistrez le à nouveau :

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Votre base de données contient les 2 versions du fichier, ainsi que le premier contenu que vous avez stocké ici :

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vous pouvez restaurer le fichier à sa première version :

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

ou à sa seconde version :

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Ce souvenir de la clé SHA-1 de chaque version de votre fichier n'est pas pratique. En plus, vous ne stockez pas le fichier lui-même, mais seulement son contenu, dans votre base. Ce type d'objets est appelé un blob (Binary Large OBject, soit en français : Gros Objet Binaire). Git peut vous donnez le type d'objet de n'importe quel objet Git, étant donné sa clé SHA-1, avec `cat-file -t` :

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

9.2.1 Objets Arbre

Le prochain type que vous allez étudier est l'objet arbre (N.d.t 'tree'), il est une solution au problème de stockage d'un groupe de fichiers. Git stocke du contenu de la même manière, mais plus simplement, qu'un système de fichier UNIX. Tout le contenu est stocké comme des objets de type arbre ou blob : un arbre correspondant à un répertoire UNIX et un blob correspond à peu près à un i-noeud ou au contenu d'un fichier. Un unique arbre contient un ou plusieurs entrées de type arbre, chacune incluant un pointeur SHA-1 vers un blob, un sous-arbre (N.d.T sub-tree), ainsi que les droits

d'accès (N.d.t 'mode'), le type et le nom de fichier. L'arbre le plus récent du projet simplegit pourrai ressembler, par exemple à ceci :

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

La syntaxe `master^tree` signifie l'objet arbre qui est pointé par le dernier commit de la branche `master`. Remarquez que le sous-répertoire `lib` n'est pas un blob, mais un pointeur vers un autre arbre :

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptuellement, les données que Git stocke ressemblent à la Figure 9.1.

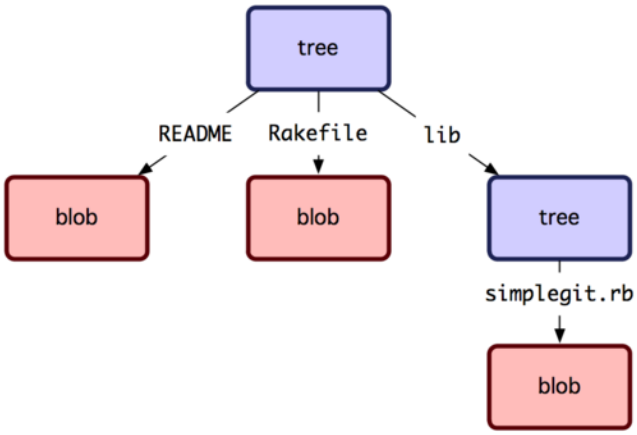

Figure 9.1: *Une version simple du modèle de données Git.*

Vous pouvez créer votre propre arbre. Git crée habituellement un arbre à partir de l'état de la zone d'attente ou de l'index. Pour créer un objet arbre, vous devez donc d'abord mettre en place un index en mettant quelques fichiers en attente. Pour créer un index contenant une entrée, la première version de votre fichier text.txt par exemple, en utilisant la commande de plomberie `update-index`. Vous pouvez utiliser cette commande pour ajouter artificiellement une version plus ancienne à une nouvelle zone d'attente. Vous devez utiliser les options `--add` car le fichier n'existe pas encore dans votre zone d'attente (vous n'avez même pas encore mis en place une zone d'attente) et `--cacheinfo` car le fichier que vous ajoutez n'est pas dans votre répertoire, mais dans la base de données. Vous pouvez ensuite préciser le mode, SHA-1, et le nom de fichier :

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Dans ce cas, vous précisez le mode `100644`, qui signifie que c'est un fichier normal. Les alternatives sont `100755`, qui signifie que c'est un exécutable et `120000`, qui précise que c'est un lien

symbolique. Le concept de « mode » a été repris des mode UNIX, mais est beaucoup moins flexible : ces trois modes sont les seuls valides pour Git, pour les fichiers (blobs) (bien que d'autres modes soient utilisés pour les répertoires et sous-modules).

Vous pouvez maintenant utiliser la commande `write-tree` pour écrire la zone d'attente dans un objet arbre. L'option' `-w` est inutile (appeler `write-tree` crée automatiquement un objet arbre à partir de l'état de l'index si cet arbre n'existe pas) :

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Vous pouvez également vérifier que c'est un objet arbre :

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Vous allez créer maintenant un nouvel arbre avec la seconde version de test.txt et un nouveau fichier :

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Votre zone d'attente contient maintenant la nouvelle version de test.txt ainsi qu'un nouveau fichier new.txt. Enregistrez cet arbre (i.e. enregistrez l'état de la zone d'attente ou de l'index dans un objet arbre) :

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Remarquez que cet arbre contient des entrées pour les deux fichiers et que l'empreinte SHA de test.txt est l'empreinte de la « version 2 » de tout à l'heure (`1f7a7a`). Pour le plaisir, ajoutez le premier arbre à celui-ci, en tant que sous-répertoire. Vous pouvez maintenant récupérer un arbre de votre zone d'attente en exécutant `read-tree`. Dans ce cas, vous pouvez récupérer un arbre existant dans votre zone d'attente comme étant un sous-arbre en utilisant l'option `--prefix` de `read-tree` :

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Si vous créez un répertoire de travail à partir du nouvel arbre que vous venez d'enregistrer, vous aurez deux fichiers à la racine du répertoire de travail, ainsi qu'un sous-répertoire appelé `bak` qui contient la première version du fichier test.txt. Vous pouvez vous représenter les données que Git utilise pour ces structures comme sur la Figure 9.2.

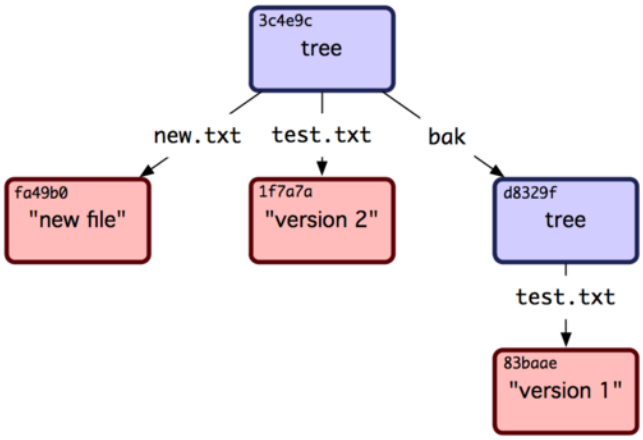

Figure 9.2: *Structure des données actuelles de Git???.*

9.2.2 Objets Commit

Vous avez trois arbres qui définissent différents instantanés du projet que vous suivez, mais certains problèmes persistent : vous devez vous souvenir des valeurs des trois empreintes SHA-1 pour accéder aux instantanés. Vous n'avez pas non plus d'information sur qui a enregistré les instantanés, quand et pourquoi. Ce sont les informations élémentaires qu'un objet commit stocke pour vous.

Pour créer un objet commit, il suffit d'exécuter `commit-tree`, de préciser l'empreinte SHA-1 et quel objet commit, s'il y en a, le précède directement. Commencez avec le premier arbre que vous avez créé :

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Vous pouvez voir votre nouvel objet commit avec `cat-file` :

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

```
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

Le format d'un commit est simple : il contient l'arbre racine de l'instantané du projet à ce moment, les informations sur l'auteur et le commiteur qui sont extraites des variables de configuration `user.name` et `user.email` accompagnées d'un horodatage, une ligne vide et le message de commit.

Ensuite, vous enregistrez les deux autres objets commit, chacun référençant le commit dont il est issu :

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Chacun des trois objets commit pointe sur un arbre de l'instantané que vous avez créez. Curieusement, vous disposez maintenant d'un historique Git complet que vous pouvez visualiser avec la commande `git log`, si vous la lancez sur le SHA-1 du dernier commit :

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  |    1 +
 test.txt |    2 +-
 2 files changed, 2 insertions(+), 1 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Fantastique. Vous venez d'effectuer les opérations bas niveaux pour construire un historique Git sans avoir utilisé aucune des commandes haut niveau. C'est l'essence de ce que fait Git quand vous exécutez les commandes `git add` et `git commit`. Il stocke les blobs correspondant aux fichiers modifiés, met à jour l'index, écrit les arbres et ajoute les objets commit qui référencent les arbres racines venant juste avant eux. Ces trois objets principaux (le blob, l'arbre et le commit) sont initialement stockés dans des fichiers séparés du répertoire `.git/objects`. Voici tous les objets contenus dans le répertoire exemple, commentés d'après leur contenu :

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Si vous suivez les pointeurs internes de ces objets, vous obtenez un graphe comme celui de la Figure 9.3.

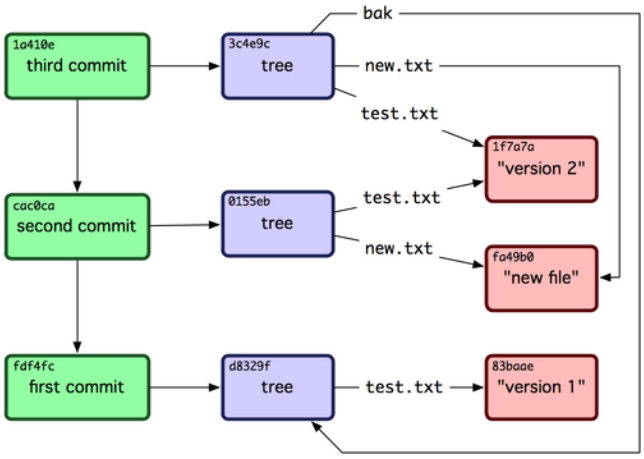

Figure 9.3: *Tous les objets de votre répertoire Git.*

9.2.3 Stockage des objets

On a parlé plus tôt de l'en-tête présent avec le contenu. Prenons un moment pour étudier la façon dont Git stocke les objets. On verra comment stocker interactivement un objet Blob (ici, la chaîne « what is up, doc? ») avec le langage Ruby. Vous pouvez démarrer Ruby en mode interactif avec la commande `irb` :

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git construit un en-tête qui commence avec le type de l'objet, ici un blob. Ensuite, il ajoute un espace suivi de taille du contenu, et enfin un octet nul :

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatène l'en-tête avec le contenu original et calcule l'empreinte SHA-1 du nouveau contenu. En Ruby, vous pouvez calculer l'empreinte SHA-1 d'une chaîne, en incluant la bibliothèque « digest/SHA-1 » via la commande `require`, puis en appelant `Digest::SHA1.hexdigest()` sur la chaîne :

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresse le nouveau contenu avec zlib, ce que vous pouvez faire avec la bibliothèque zlib de Ruby. Vous devez inclure la bibliothèque et exécuter `Zlib::Deflate.deflate()` sur le contenu :

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

Finalement, vous enregistrerez le contenu compressé dans un objet sur le disque. Vous déterminerez le chemin de l'objet que vous voulez enregistrer (les deux premiers caractères de l'empreinte SHA-1 formeront le nom du sous-répertoires, et les 38 derniers formeront le nom du fichier dans ce répertoire). En Ruby, on peut utiliser la fonction `FileUtils.mkdir_p()` pour créer un sous-répertoire s'il n'existe pas. Ensuite, ouvrez le fichier avec `File.open()` et enregistrez le contenu compressé en appelant la fonction `write()` sur la référence du fichier :

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

C'est tout ! Vous venez juste de créer un objet Blob valide. Tout les objets Git sont stockés de la même façon, mais avec des types différents : l'en-tête commencera par « commit » ou « arbre »

au lieu de la chaîne « blob ». Bien que le contenu d'un blob puisse être presque n'importe quoi, le contenu d'un commit ou d'un arbre est formaté d'une façon particulière.

9.3 Références Git

On peut exécuter quelque chose comme `git log 1a410e` pour visualiser tout l'historique, mais il faut se souvenir que `1a410e` est le dernier commit afin de parcourir l'historique et trouver tous ces objets. Vous avez besoin d'un fichier ayant un nom simple qui contient l'empreinte SHA-1 afin d'utiliser ce pointeur plutôt que l'empreinte SHA-1 elle-même.

Git appelle ces pointeur des « références », ou « refs ». On trouve les fichiers contenant des empreintes SHA-1 dans le répertoire `git/refs`. Dans le projet actuel, ce répertoire ne contient aucun fichier, mais possède une structure simple :

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Pour créer une nouvelle référence servant à ce souvenir du dernier commit, vous pouvez simplement faire ceci :

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Vous pouvez maintenant utiliser la référence principale que vous venez de créer à la place de l'empreinte SHA-1 dans vos commandes Git :

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Il n'est pas conseillé d'éditer directement les fichiers des références. Git propose une manière sûre de mettre à jour une référence, c'est la commande `update-ref` :

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

C'est simplement ce qu'est une branche dans Git : un simple pointeur ou référence sur le dernier état d'une suite de travaux. Pour créer une branche à partir du deuxième commit, vous pouvez faire ceci :

```
$ git update-ref refs/heads/test cac0ca
```

Cette branche contiendra seulement le travail effectué jusqu'à ce commit :

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

La base de donnée Git ressemble maintenant à quelque chose comme la Figure 9.4.

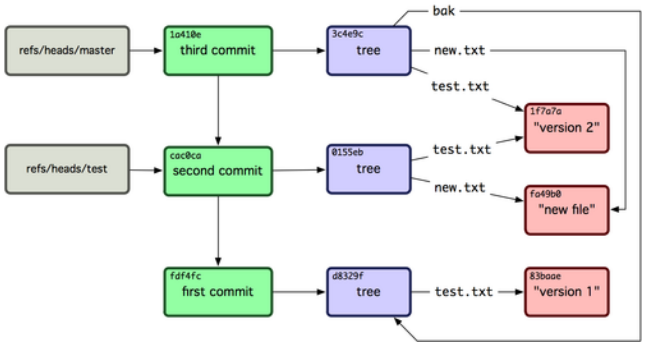

Figure 9.4: *Le répertoire d'objet de Git y compris la référence au dernier état de la branche.*

Quand on exécute un commande comme `git branch (nomdebranche)`, Git exécute simplement la commande `update-ref` pour ajouter l'empreinte SHA-1 du dernier commit dans la référence que l'on veut créer.

9.3.1 La branche HEAD

On peut se poser la question : Comment Git peut avoir connaissance de l'empreinte SHA-1 du dernier commit quand on exécute `git branch (branchname)` ? La réponse est dans le fichier HEAD (qui veut dire tête en français, soit, ici, l'état courant). Le fichier HEAD est une référence symbolique à la branche courante. Par référence symbolique, j'entends que contrairement à une référence normale, elle ne contient pas une empreinte SHA-1, mais plutôt un pointeur vers une autre référence. Si vous regardez ce fichier, vous devriez voir quelque chose comme ceci :

```
$ cat .git/HEAD
ref: refs/heads/master
```

Si vous exécutez `git checkout test`, Git met à jour ce fichier, qui ressemblera à ceci :

```
$ cat .git/HEAD
ref: refs/heads/test
```

Quand vous exécutez `git commit`, il crée l'objet commit en spécifiant le parent du commit comme étant l'empreinte SHA-1 pointé par la référence du fichier HEAD :

On peut éditer manuellement ce fichier, mais encore une fois, il existe une commande supplémentaire pour le faire : `symbolic-ref`. Vous pouvez lire le contenu de votre fichier HEAD avec cette commande :

```
$ git symbolic-ref HEAD
refs/heads/master
```

Vous pouvez aussi initialiser la valeur de HEAD :

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Vous ne pouvez pas initialiser une référence symbolique à une valeur non contenu dans refs :

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

9.3.2 Étiquettes

Nous venons de parcourir les trois types d'objet utilisé par Git, mais il existe un quatrième objet. L'objet étiquette (tag en anglais) ressemble beaucoup à un objet commit. Il contient un étiqueteur, une date, un message, et un pointeur. La principale différence est que l'étiquette pointe vers un commit plutôt qu'un arbre. C'est comme une référence à une branche, mais elle ne bouge jamais : elle pointe toujours vers le même commit, lui donnant un nom plus sympathique.

Comme présenté au chapitre 2, il existe deux types d'étiquettes : annotée et légère. Vous pouvez créer une étiquette légère comme ceci :

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

C'est tout ce qu'est une étiquette légère : une branche qui n'est jamais modifiée. Une étiquette annotée annoté est plus complexe. Quand on crée une étiquette annotée, Git crée un objet étiquette, puis enregistre une référence qui pointe vers lui plutôt que directement vers le commit. Vous pouvez voir ceci en créant une étiquette annotée (`-a` spécifie que c'est une étiquette annotée) :

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Voici l'empreinte SHA-1 de l'objet créé :

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Exécutez ensuite, la commande `cat-file` sur l'empreinte SHA-1 :

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Remarquez que le contenu de l'objet pointe vers l'empreinte SHA-1 du commit que vous avez étiqueté. Remarquez qu'il n'est pas nécessaire qu'il pointe vers un commit. On peut étiqueter n'importe quel objet. Par exemple, dans le code source de Git, le mainteneur a ajouté ses clé GPG dans un blob et l'a étiqueté. Vous pouvez voir la clé publique en exécutant :

```
$ git cat-file blob junio-gpg-pub
```

dans le code source de Git. Le noyau linux contient aussi une étiquette ne pointant pas vers un commit : la première étiquette créée pointe vers l'arbre initial lors de l'importation du code source.

9.3.3 Références distantes

Le troisième type de références que l'on étudiera sont les références distantes (N.d.T remotes). Si l'on ajoute une référence distante et que l'on pousse des objets vers elle, Git stocke la valeur que vous avez poussé en dernière vers cette référence pour chaque branche dans le répertoire `refs/remotes`. Vous pouvez par exemple, ajouter une référence distante nommée `origin` et y pousser votre branche `master` :

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
   a11bef0..ca82a6d  master -> master
```

Ensuite, vous pouvez voir l'état de la branche `master` dans la référence distante `origin` la dernière fois que vous avez communiqué avec le serveur en regardant le fichier `refs/remotes/origin/master` :

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Les références distantes diffèrent des branches (références `refs/heads`) principalement parce qu'on ne peut pas les récupérer dans le répertoire de travail. Git les modifie comme des marque pages du dernier état de ces branches sur le serveur.

9.4 Fichiers groupés

Revenons à la base de donnée d'objet de notre dépôt Git de test. Pour l'instant, il contient 11 objets : 4 blobs, 3 arbres, 3 commits, et 1 tag :

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # arbre 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # arbre 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # arbre 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresse le contenu de ces fichiers avec zlib, et on ne stocke pas grand chose, au final, tous ces fichiers occupent seulement 925 octets. Ajoutons de plus gros contenu au dépôt pour montrer une fonctionnalité intéressante de Git. Ajoutez le fichier repo.rb de la bibliothèque Grit que vous avez manipuler plus tôt. Il représente environ 12Ko de code source :

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Si vous observez l'arbre qui en résulte, vous verrez l'empreinte SHA-1 du blob contenant le fichier repo.rb :

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Vous pouvez utilisez `git cat-file` pour connaitre la taille de l'objet :

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Maintenant, modifiez le fichier un peu, et voyez ce qui arrive :

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Regardez l'arbre créé par ce commit, et vous verrez quelque chose d'intéressant :

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Ce blob est un blob différent. Bien que l'on ait ajouté une seule ligne à la fin d'un fichier en faisant 400, Git enregistre ce nouveau contenu dans un objet totalement différent :

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
12908
```

Il y a donc deux objets de 12Ko quasiment identique sur le disque. Ne serait-ce pas bien si Git pouvait enregistrer un objet en entier, et le deuxième n'étant qu'un delta (une différence) avec le premier ?

Il se trouve que c'est possible. Le format initial dans lequel Git enregistre les objets sur le disque est appelé le format brut (« loose object »). De temps en temps, Git compacte plusieurs de ces objets en un seul fichier binaire appelé packfile (fichier groupé), afin d'économiser de l'espace et d'être plus efficace. Git effectue cette opération quand il y a trop d'objets au format brut, ou si l'on exécute manuellement la commande `git gc`, ou encore quand on pousse vers un serveur distant. Pour voir cela en action, vous pouvez demander manuellement à Git de compacter les objets en exécutant la commande `git gc` :

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Si l'on jette un œil dans le répertoire des objets, on constatera que la plupart des objets ne sont plus là et qu'un couple de fichier est apparu :

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Les objets restant sont des blobs qui ne sont pointés par aucun commit. Dans notre cas, il s'agit des blobs « what is up, doc? » et « test content » créés plus tôt comme exemple. Puisqu'il n'ont été ajouté à aucun commit, ils sont considérés en suspend et ne sont pas compacter dans le nouveau fichier groupé.

Les autres fichiers sont le nouveau fichier gorupé et un index. Le fichier groupé est un fichier unique contenant le contenu de tous les objets venant d'être supprimés du système de fichier. L'index est un fichier contenant les emplacements dans le fichier groupé, pour que l'on puisse accéder rapidement à un objet particulier. Ce qui est vraiment bien est que les objets occupaient environ 12Ko d'espace disque avant `gc`, et que le nouveau fichier groupé en occupe seulement 6Ko. On a divisé par deux l'occupation du disque en regroupant les objets.

Comment Git réalise-t-il cela ? Quand Git compacte des objets, il recherche les fichiers qui ont des noms et des tailles similaires, puis enregistre seulement les deltas entre une version du fichier et la suivante. On peut regarder à l'intérieur du fichier groupé et voir l'espace économisé par Git. La commande de plomberie `git verify-pack` vous permet de voir ce qui a été compacté :

```
$ git verify-pack -v \
  .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 5400
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree   106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag    136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob   7 18 5193 1
05408d195263d853f09dca71d55116663690c27c \
  ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree   106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Si on se souvient bien, le blob `9bc1d`, qui est la première version de fichier repo.rb file,

référence le blob `05408`, qui est la seconde version du fichier. La troisième colonne de l'affichage est la taille de l'objet dans le fichier compact, et on peut voir que `05408` occupe 12Ko dans le fichier, mais que `9bc1d` occupe seulement 7 octets. Ce qui est aussi intéressant est que la seconde version du fichier est celle qui est enregistrée telle quelle, tandis que la version originale est enregistrée sous forme d'un delta. La raison en est que vous aurez sans doute besoin d'accéder rapidement aux versions les plus récentes du fichier.

Une chose intéressante à propos de ceci est que l'on peut recompacter à tout moment. Git recompacte votre base de donnée occasionnellement, en essayant d'économiser de la place. Vous pouvez aussi recompacter à la main, en exécutant la commande `git gc` vous-même.

9.5 Les références spécifiques

Dans tout le livre, nous avons utilisé des associations simples entre les branches distantes et les références locales. Elles peuvent être plus complexes. Supposons que vous ajoutiez un dépôt distant comme ceci :

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Cela ajoute une section au fichier `.git/config`, contenant le nom du dépôt distant (`origin`), l'URL de ce dépôt, et la spécification des références pour la récupération :

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
```

Le format d'une spécification de référence est un + facultatif, suivi de `<src>:<dst>`, où `<src>` est le motif des références du côté distant, et `<dst>` est l'emplacement local où les références seront enregistrées. Le + précise à Git de mettre à jour la référence même si ce n'est pas un avance rapide.

Dans le cas par défaut, qui est celui d'un enregistrement automatique par la commande `git remote add`, Git récupère toutes les références de `refs/heads/` sur le serveur et les enregistre localement dans `refs/remotes/origin/`. Ainsi, s'il y a une branche `master` sur le serveur, vous pouvez accéder localement à l'historique de cette branche via :

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Ces syntaxes sont toutes équivalentes, car Git les développe en `refs/remotes/origin/master`.

Si vous préférez que Git récupère seulement la branche `master`, et non chacune des branches du serveur distant, vous pouvez remplacer la ligne fetch par :

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

C'est la spécification des références de `git fetch` pour ce dépôt distant. Si l'on veut effectuer une action particulière une seule fois, la spécification des références peut aussi être précisée en ligne de commande. Pour retirer la branche `master` du dépôt distant vers la branche locale `origin/mymaster`, vous pouvez exécutez :

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Vous pouvez indiquer des spécifications pour plusieurs références. En ligne de commande, vous pouvez tirer plusieurs branches de cette façon :

```
$ git fetch origin master:refs/remotes/origin/mymaster \
   topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

Dans ce cas, la récupération /* pull */ de la branche `master` a été refuséé car ce n'était pas une avance rapide. On peut surcharger ce comportement en précisant un + devant la spécification de la référence.

On peut aussi indiquer plusieurs spécifications de référence pour la récupération, dans le fichier de configuration. Si vous voulez toujours récupérer les branches `master` /*manque les backquotes dans la version en*/ et `experiment`, /* idem*/ ajoutez ces deux lignes :

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Vous ne pouvez pas utiliser des jokers partiels, ce qui suit est donc invalide :

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

On peut toutefois utiliser des espaces de noms pour accomplir cela. S'il existe une équipe qualité (QA) qui publie une série de branches, et que l'on veut la branche `master`, les branches de l'équipe qualité et rien d'autre, on peut utiliser la configuration suivante :

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Si vous utilisez des processus complexes impliquant un équipe qualité, des développeurs, et des intégrateurs qui publient des branches et qui collaborent sur des branches distantes, vous pouvez facilement utiliser des espaces des noms de cette façon.

9.5.1 Publier une référence spécifique

Il est pratique de pouvoir récupérer des références issues d'espace de nom de cette façon, mais comment l'équipe qualité insère-t-elle ces branches dans l'espace de nom `qa/` en premier lieu ? On peut accomplir cela en utilisant les spécification de références pour la publication.

Si l'équipe qualité veut publier sa branche `master` vers `qa/master` sur le serveur distant, elle peut exécuter :

```
$ git push origin master:refs/heads/qa/master
```

Si elle veut que Git le fasse automatiquement à chaque exécution de `git push origin`, elle peut ajouter une entrée `push` au fichier de configuration :

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
       push = refs/heads/master:refs/heads/qa/master
```

De même, cela fera que, par défaut, `git push origin` publiera la branche locale `master` sur la branche distante `qa/master`.

9.5.2 Supprimer des références

Vous pouvez aussi utiliser les spécifications de références pour supprimer des références sur le serveur distant en exécutant une commande comme :

```
$ git push origin :topic
```

La spécification de référence ressemble à `<src>:<dst>`, mais en laissant vide la partie `<src>`, cela signifie de créer la branche à partir de rien, et donc la supprime.

9.6 Protocoles de transfert

Git peut transférer des données entre deux dépôts, de deux façons principales : via HTTP et via un protocole dit « intelligent » utilisé par les transports /*trouver mieux*/ `file://`, `ssh://` et

`git://`. Cette section fait un tour d'horizon du fonctionnement de ces deux protocoles.

9.6.1 Le protocole stupide

On parle souvent du transfert Git sur HTTP comme étant un protocole stupide, car il ne nécessite aucun code spécifique à Git côté serveur durant le transfert. Le processus de récupération est une série de requête GET, où le client devine la structure du dépôt Git présent sur le serveur. Suivons le processus `http-fetch` pour la bibliothèque simplegit :

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

La première chose que fait cette commande est de récupérer le fichier `info/refs`. Ce fichier est écrit par la commande `update-server-info`, et c'est pour cela qu'il faut activer le hook `post-receive`, sinon le transfert HTTP ne fonctionnera pas correctement :

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

On possède maintenant une liste des références distantes et empreintes SHA1. Ensuite, on regarde vers quoi pointe HEAD, pour savoir sur quelle branche se placer quand on aura fini :

```
=> GET HEAD
ref: refs/heads/master
```

On aura besoin de se placer sur la branche `master`, quand le processus sera terminé. On est maintenant prêt à démarrer le processus de parcours. Puisque votre point de départ est l'objet commit `ca82a6` que vous avez vu dans le fichier `info/refs`, vous commencez par le récupérer :

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Vous obtenez un objet, cet objet est dans le format brut sur le serveur, et vous l'avez récupéré à travers une requête HTTP GET statique. Vous pouvez le décompresser avec zlib, ignorer l'entête, et regarder le contenu du commit :

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Puis, vous avez deux autres objets supplémentaires à récupérer : `cfda3b` qui est l'arbre du contenu sur lequel pointe le commit que nous venons de récupérer, et `085bb3` qui est le commit parent :

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Cela vous donne l'objet du prochain commit. Récupérez l'objet arbre :

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oups, on dirait que l'objet arbre n'est pas au format brut sur le serveur, vous obtenez donc une réponse 404. On peut en déduire certaines raisons : l'objet peut être dans un dépôt suppléant, ou il peut être dans un fichier groupé de ce dépôt. Git vérifie la liste des dépôts suppléants d'abord :

```
=> GET objects/info/http-alternates
(empty file)
```

Si la réponse contenait une liste d'URLs suppléantes, Git aurait cherché les fichiers bruts et les fichiers groupés à ces emplacements, c'est un mécanisme sympatique pour les projets qui ont dérivés d'un autre pour partager les objets sur le disque. Cependant, puisqu'il n'y a pas de suppléants listés dans ce cas, votre objet doit se trouver dans un fichier groupé. Pour voir quels fichiers groupés sont disponibles sur le serveur, vous avez besoin de récupérer le fichier `objects/info/packs`, qui en contient la liste (générée également par `update-server-info`) :

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Il n'existe qu'un seul fichier groupé sur le serveur, votre objet se trouve évidemment dedans, but vous allez tout de même vérifier l'index pour être sûr. C'est également utile lorsque vous avez plusieurs fichiers groupés sur le serveur, vous pouvez donc voir quel fichier groupé contient l'objet que vous avez besoin :

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Maintenant que vous avez l'index du fichier groupé, vous pouvez voir si votre objet est bien dedans, car l'index liste les empreintes SHA-1 des objets contenus dans ce fichier groupé et des emplacements de ces objets. Votre objet est là, allez donc récupérer le fichier groupé complet :

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Vous avez votre objet arbre, vous continuez donc le chemin des commits. Ils sont également tous contenus dans votre fichier groupé que vous venez de téléchargé, vous n'avez donc pas d'autres requêtes à faire au serveur. Git récupère un copie de travail de votre branche `master` qui été référencé par HEAD que vous avez téléchargé au début.

La sortie complète de cette procédure ressemble à :

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 Le Protocole Intelligent

La méthode HTTP est simple mais un peu inefficace. Utiliser des protocoles intelligents est une méhode plus habituelles pour transférer des données. Ces protocoles ont un exécutable du côté distant qui connait Git, il peut lire les données locales et deviner ce que le client a ou ce qu'il a besoin pour générer des données personnalisées pour lui. Il y a deux ensembles d'exécutables pour transférer les données : une paire pour téléverser des données et une paire pour en télécharger.

Téléverser des données

Pour téléverser des données vers un exécutable distant, Git utilise les exécutables `send-pack` et `receive-pack`. L'exécutable `send-pack` tourne sur le client et se connecte à l'exécutable `receive-pack` du côté serveur.

Par exemple, disons que vous exécutez `git push origin master` dans votre projet, et `origin` est défini comme une URL qui utilise le protocole SSH. Git appelle l'exécutable `send-pack`, qui initialise une connexion à travers SSH vers votre serveur. Il essaye d'exécuter une commande sur le serveur distant via un appel SSH qui ressemble à :

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

La commande `git-receive-pack` répond immédiatement avec une ligne pour chaque référence qu'elle connait actuellement, dans ce cas, uniquement la branche `master` et ses empreintes SHA. La première ligne contient également une liste des compétences du serveur (ici : `report-status` et `delete-refs`).

Chaque ligne commence avec une valeur hexadécimale sur 4 octets, spécifiant le reste de la longueur de la ligne. La première ligne, ici, commence avec `005b`, soit 91 en hexadécimal, ce qui signifie qu'il y a 91 octets restants sur cette ligne La ligne suivante commence avec `003e`, soit 62, vous lisez donc les 62 octets restants. La ligne d'après est `0000`, signifiant que le serveur a fini de lister ses références.

Maintenant que vous connaissez l'état du serveur, votre exécutable `send-pack` détermine quels commits il a que le serveur n'a pas. L'exécutable `send-pack` envoit alors à l'exécutable `receive-pack`, les informations concernant chaque référence que cette commande `push` va mettre à jour. Par exemple, si vous mettez à jour la branche `master` et ajoutez la branche `experiment`, la réponse de `send-pack` ressemblera à quelque chose comme :

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 refs/
heads/master report-status
006700000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/
heads/experiment
0000
```

La valeur SHA-1 remplie de '0' signifie qu'il n'y avait rien à cet endroit avant, car vous êtes en train d'ajouter la référence `experiment`. Si vous êtiez en train de supprimer une référence, vous verriez l'opposé : que des '0' du côté droit.

Git envoit une ligne pour chaque référence que l'on met à jour avec l'ancien SHA, le nouveau SHA, et la référence en train d'être mise à jour. La première ligne contient également les compétences du client. Puis, le client téléverse un fichier groupé de tous les objets que le serveur n'a pas encore. Finalement, le serveur répond avec une indication de succès (ou d'échec) :

```
000Aunpack ok
```

Télécharger des Données

Lorsque vous télécharger des données, les exécutables `fetch-pack` et `upload-pack` entrent en jeu. Le client initialise un exécutable `fetch-pack` qui se connecte à un exécutable `upload-pack` du côté serveur pour négocier quelles données seront remontées.

Il y a plusieurs manières d'initialiser l'exécutable `upload-pack` sur le dépôt distant. Vous pouvez passer par SSH de la même manière qu'avec l'exécutable `receive-pack`. Vous pouvez également initialiser l'exécutable à travers le démon Git, qui écoute sur le port 9418 du serveur par défaut. L'exécutable `fetch-pack` envoit des données qui ressemble à cela juste après la connexion :

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Cela commence par les 4 octets désignant la quantité de données qui suit, puis la commande à exécuter suivi par un octet nul, puis le nom d'hôte du serveur suivi d'un octet nul final. Le démon Git vérifie que la commande peut être exécutée, que le dépôt existe et soit accessible publiquement. Si tout va bien, il appelle l'exécutable `upload-pack` et lui passe la main.

Si vous êtes en train de tirer (fetch) à travers SSH, `fetch-pack` exécute plutôt quelque chose du genre :

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

Dans tous les cas, après que `fetch-pack` se connecte, `upload-pack` lui répond quelque chose du style :

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

C'est très proche de ce que répondait `receive-pack`, mais les compétences sont différentes. En plus, il vous répond la référence HEAD, afin que le client sache quoi récupérez dans le cas d'un clone.

À ce moment, l'exécutable `fetch-pack` regarde quels objets il a et répond avec les objets dont il a besoin en envoyant « want » (vouloir) suivi du SHA qu'il veut. Il envoit tous les objets qu'il a déjà avec « have » suivi du SHA. À la fin de la liste, il écrit « done » pour initialiser l'exécutable `upload-pack` à commencer à envoyer le fichier groupé des données demandées :

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

C'est le cas basique d'un protocole de transfert. Dans des cas plus complexes, le client a des compétendes `multi_ack` (plusieurs réponses) ou `side-band` (plusieurs connexions), mais cet exemple vous montre les bases du protocole intelligent.

9.7 Maintenance et Récupération de Données

Parfois, vous aurez besoin de faire un peu de ménage : faire un dépôt plus compact, nettoyer les dépôts importés, ou récupérer du travail perdu. Cette section couvrira certains de ces scenarii.

9.7.1 Maintenance

De temps en temps, Git exécute automatiquement une commande appelée « auto gc ». La plupart du temps, cette commande ne fait rien. Cependant, s'il y a trop d'objets bruts (des objets qui

ne sont pas dans des fichiers groupés), ou trop de fichiers groupés, Git lance une commande `git gc` à part entière. `gc` est l'abbréviation pour « garbage collect » (ramasse-miettes), et la commande fait plusieurs choses : elle rassemble plusieurs objets bruts et les place dans un fichiers groupés, elle consolide des fichiers groupés en un gros fichier groupé, et elle supprim des objets qui ne sont plus accessibles depuis un commit et qui sont vieux de plusieurs mois.

Vous pouvez exécuter `auto gc` manuellement :

```
$ git gc --auto
```

Encore une fois, cela ne fait généralement rien. Vous devez avoir environ 7.000 objets bruts ou plus de 50 fichiers groupés pour que Git appelle une vraie commande `gc`. Vous pouvez modifier ces limites avec les propriétés de configuration `gc.auto` et `gc.autopacklimit`, respectivement.

`gc` regroupera aussi vos références dans un seul fichier. Supposons que votre dépôt contienne les branches et étiquettes suivantes :

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Si vous exécutez `git gc`, vous n'aurez plus ces fichiers dans votre répertoire `refs`. Git les déplacera pour le bien de l'efficacité dans un fichier nommé `.git/packed-refs` qui ressemble à ceci :

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Si vous mettez à jour une référence, Git ne modifiera pas ce fichier, mais enregistrera plutôt un nouveau fichier dans `refs/heads`. Pour obtenire l'empreinte SHA approprié pour référence donnée, Git cherche d'abord cette référence dans le répertoire `refs`, puis dans le fichier `packed-refs` si non trouvée. Cependant, vous ne pouvez pas trouver une référence dans votre répertoire `refs`, elle est probablement dans votre fichier `packed-refs`.

Remarquez la dernière ligne du fichier, celle commençant par `^`. Cela signifie que l'étiquette directement au-dessus est une étiquette annotée et que cette ligne est le commit que l'étiquette annotée référence.

9.7.2 Récupération de données

À un moment quelconque de votre vie avec Git, vous pouvez accentillement perdre un commit. Généralement, cela arrive parce que vous avez forcé la suppression d'une branche contenant du tra-

vail, et il se trouve que vous voulez cette branche finalement; ou vous avez réinitialiser une branche avec suppression, en abandonnant des commits dont vous vouliez des informations. Supposons que cela arrive, comment pouvez-vous récupérer vos commits ?

Voici un exemple qui réinitialise la branche `master` avec suppression dans votre dépôt de test vers un ancien commit et qui récupère les commits perdus. Premièrement, vérifions dans quel état est votre dépôt en ce moment :

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Maintenant, déplaçons la branche `master` vers le commit du milieu :

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Vous avez effectivement perdu les deux commits du haut, vous n'avez pas de branche depuis laquelle ces commits seraient accessibles. Vous avez besoin de trouver le SHA du dernier commit et d'ajouter une branche s'y référant. Le problème est de trouver ce SHA, ce n'est pas comme si vous l'aviez mémorisé, hein ?

Souvent, la manière la plus rapide est d'utiliser l'outil `git reflog` Pendant que vous travaillez, Git enregistre l'emplacement de votre HEAD chaque fois que vous le changez. À chaque commit ou commutation de branche, le journal des références (reflog) est mis à jour. Le journal des références est aussi mis à jour par la commande `git update-ref`, qui est une autre raison de l'utiliser plutôt que de simplement écrire votre valeur SHA dans vos fichiers de références, comme mentionné dans la section « Git References » plus haut dans ce chapitre. Vous pouvez voir où vous étiez à n'importe quel moment en exécutant `git reflog` :

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Ici, nous pouvons voir deux commits que nous avons récupérés, cependant, il n'y a pas plus d'information ici. Pour voir, les mêmes informations d'une manière plus utile, nous pouvons exécuter `git log -g`, qui nous donnera une sortie normalisée pour votre journal de références :

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

     modified repo a bit
```

On dirait que le commit du bas est celui que vous avez perdu, vous pouvez donc le récupérer en créant une nouvelle branche sur ce commit. Par exemple, vous créez une branche nommée `recover-branch` au commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Super, maintenant vous avez une nouvelle branche appelée `recover-branch` à l'emplacement où votre branche `master` était, faisant en sorte que les deux premiers commits soit à nouveau accessibles.

Pour poursuivre, nous supposerons que vos pertes ne sont pas dans le journal des références pour une raison quelconque. On peut simuler cela en supprimant `recover-branch` et le journal des références. Maintenant, les deux premiers commits ne sont plus accessibles (encore) :

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Puisque les données du journal de référence sont sauvegardés dans le répertoire `.git/logs/`, vous n'avez effectivement plus de journal de références. Comment pouvez-vous récupérer ces commits maintenant ? Une manière de faire est d'utiliser l'outil `git fsck`, qui vérifie l'intégrité de votre base de données. Si vous l'exécutez avec l'option `--full`, il vous montre tous les objets qui ne sont pas référencés par d'autres objets :

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Dans ce cas, vous pouvez voir votre commit manquant après « dangling commit ». Vous pouvez le restaurez de la même manière que précédemment, en créant une branche qui référence cette empreinte SHA.

9.7.3 Supprimer des objets

Il y a beaucoup de choses dans Git qui sont géniales, mais une fonctionnalité qui peut poser problème est le fait que `git clone` télécharge l'historique entier du projet, incluant chaque version de chaque fichier. C'est très bien lorsque le tout est du code source, parce Git est hautement optimisé pour compresser les données efficacement. Cependant, si quelqu'un à un moment donné de l'historique de votre projet a ajouté un énome fichier, chaque clone sera forcé de télécharger cet énorme fichier, même si il a été supprimé du projet dans le commit suivant. Puisqu'il est accessible depuis l'historique, il sera toujours là.

Cela peut être un énorme problème, lorsque vous convertissez un dépôt Subversion ou Perfoce en un dépôt Git. Car, comme vous ne téléchargez pas l'historique entier dans ces syst-mes, ce genre d'ajout n'a que peu de conséquences. Si vous avez importé depuis un autre système ou que votre dépôt est beaucoup plus gros que ce qu'il devrait être, voici comment vous pouvez trouver et supprimer des gros objets.

Soyez prévenu : cette technique détruit votre historique de commit. Elle réécrit chaque objet commit depuis le premier objet arbre que vous modifiez pour supprimer une référence d'un gros fichier. Si vous faites cela immédiatement après un import, avant que quiconque n'ait eu le temps de commencer à travailler sur ce commit, tout va bien. Sinon, vous devez alerter tous les contributeurs qu'ils doivent recommencer (ou au moins faire un `rebase`) sur votre nouveau commit.

Pour la démonstration, nous allons ajouter un gros fichier dans votre dépôt de test, le supprimer dans le commit suivant, le trouver, et le supprimer de manière permanente du dépôt. Premièrement, ajoutons un gros objet à votre historique :

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

Oups, vous ne vouliez pas rajouter une énorme archive à votre projet. Il vaut mieux s'en débarrasser :

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Maintenant, faites un `gc` sur votre base de données, pour voir combien d'espace disque vous utilisez :

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Vous pouvez exécutez la commande `count-objects` pour voir rapidement combien d'espace disque vous utilisez :

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

L'entrée `size-pack` est la taille de vos fichiers groupés en kilo-octets, vous utilisez donc 2Mo. Avant votre dernier commit, vous utilisiez environ 2Ko, clairement, supprimer le fichier avec le commit précédent, ne l'a pas enlever de votre historique. À chaque fois que quelqu'un clonera votre dépôt, il aura à cloner les 2Mo pour récupérer votre tout petit projet, parce que vous avez accidentellement rajouté un gros fichier. Débarrassons-nous en.

Premièrement, vous devez le trouver. Dans ce cas, vous savez déjà de quel fichier il s'agit. Mais supposons que vous ne le sachiez pas, comment identifieriez-vous quel(s) fichier(s) prennent trop de place ? Si vous exécutez `git gc`, tous les objets sont des dans fichiers groupés; vous pouvez identifier les gros objets en utilisant une autre commande de plomberie appelée `git verify-pack` et en triant sur le troisième champ de la sortie qui est la taille des fichiers. Vous pouvez également la faire suivre à la commande `tail` car vous ne vous intéressez qu'aux fichiers les plus gros :

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob   2056716 2056872 5401
```

Le gros objet est à la fin : 2Mo. Pour trouver quel fichier c'est, vous allez utilisez la commande `rev-list`, que vous avez utilisé brièvement dans le chapitre 7. Si vous mettez l'option `--objects` à `rev-list`, elle listera tous les SHA des commits et des blobs avec le chemin du fichier associés. Vous pouvez utilisez cette commande pour trouver le nom de votre blob :

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Maintenant, vous voulez supprimer ce fichier de toutes les arborescences passées. Vous pouvez facilement voir quels commits ont modifié ce fichier :

```
$ git log --pretty=oneline -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Vous devez réécrire tous les commits qui sont liés à `6df76` pour supprimer totalement ce fichier depuis votre historique Git. Pour cela, utilisez `filter-branch`, que vous avez utilisé dans le Chapitre 6 :

```
$ git filter-branch --index-filter \
   'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

L'option `--index-filter` est similaire à l'option `--tree-filter` utilisé dans le Chapitre 6, sauf qu'au lieu de modifier les fichiers sur le disk, vous modifiez votre zone d'attente et votre index. Plutôt que de supprimer un fichier spécifique avec une commande comme `rm file`, vous devez le supprimez avec `git rm --cached`; vous devez le supprimer de l'index, pas du disque. La raison de faire cela de cette manière est la rapidité, car Git n'ayant pas besoin de récupérer chaque révision sur disque avant votre filtre, la procédure peut être beaucoup beaucoup plus rapide. Vous pouvez faire la même chose avec `--tree-filter` si vous voulez. L'option `--ignore-unmatch` de `git rm` lui dit que ce n'est pas une erreur si le motif que vous voulez supprimez n'existe pas. Finalement, vous demandez à `filter-branch` de réécrire votre historique seulement depuis le parent du commit `6df7640`, car vous savez que c'est de là que le problème a commencé. Sinon, il aurait démarré du début, et serait plus long sans nécessité.

Votre historique ne contient plus de référence à ce fichier. Cependant, votre journal de révision et un nouvel ensemble de référence que Git a ajouté lors de votre `filter-branch` dans `.git/refs/original` en contiennent encore, vous devez donc les supprimer puis regrouper votre base de données. Vous devez vous débarrasser de tout ce qui se référence à ces vieux commits avant de regrouper :

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Voyons combien d'espace vous avez récupéré :

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

La taille du dépôt regroupé est retombé à 7Ko, ce qui est beaucoup que 2Mo. Vous pouvez voir dans la valeur « size » que votre gros objet est toujours dans vos objets bruts, il n'est donc pas parti; mais il ne sera plus transféré lors d'une poussée vers un serveur ou un clone, ce qui est l'important dans l'histoire. Si vous voulez réellement, vous pouvez supprimer complètement l'objet en exécutant `git prune --expire`.

9.8 Résumé

Vous devriez avoir une plutôt bonne compréhension de ce que Git fait en arrière plan, et, à un certain degré, comment c'est implémenté. Ce chapitre a parcouru un certain nombre de commande de plomberie, commandes qui sont à un niveau plus bas et plus simple que les commandes de porcelaine que vous avez après dans le reste du livre. Comprendre comment Git travaille à bas niveau devrait vous aider à comprendre pourquoi il fait ce qu'il fait et à créer vos propres outils et scripts pour que votre workflow fonctionne comme vous l'entendez.

Git, comme un système de fichiers adressables par contenu, est un outil puissant que vous pouvez utilisez pour des fonctionnalités au delà d'un SVC. J'espère que vous pourrez utilisez votre connaissance nouvellement acquise des trippes de Git pour implémenter votre propre super application de cette technologique et que vous vous sentirez plus à l'aise à utiliser Git de manière plus poussée.